

Joseph Eremondi*, Wouter Swierstra, and Jurriaan Hage

A Framework for Improving Error Messages in Dependently Typed Languages

<https://doi.org/DOI>

Received ..; revised ..; accepted ..

Abstract: Dependently typed programming languages provide a powerful tool for establishing code correctness. However, it can be hard for newcomers to learn how to employ the advanced type system of such languages effectively. For simply-typed languages, several techniques have been devised to generate helpful error messages and suggestions for the programmer. We adapt these techniques to dependently typed languages, to facilitate their more widespread adoption. In particular, we modify a higher order unification algorithm that is used to resolve and typecheck implicit arguments. We augment this algorithm with *replay graphs*, allowing for a global heuristic analysis of a unification problem-set, *error-tolerant typing*, which allows typechecking to continue after errors are found, and *counter-factual unification*, which makes error messages less affected by the order in which types are checked. A formalization of our algorithm is presented with an outline of its correctness. We implement the algorithm and compare its error messages to those from existing languages, highlighting the improvements we achieve.

Keywords: higher order unification, type error diagnosis, counter-factual typing, type inference

1 Introduction

Dependent types enable developers to enforce rich properties of their programs statically. Through the Curry-Howard Correspondence, a dependent type system provides the full power of higher-order logic for proving program correctness.

Notwithstanding, dependent types have not seen widespread adoption. This may be in part due to their experimental nature, as they are still an active subject of research. The type systems that allow them to provide so many guarantees also enforce a very strict type discipline

when programming, and this complexity leads to more complex error messages reported to the programmer. Thus, a large amount of development time is spent responding to error messages, diagnosing errors, and repairing them.

For beginning programmers, these type errors can prove a daunting barrier to entry, particularly when the error messages that the compiler provides are uninformative or confusing. Empirical evidence suggests that difficulty reading error messages is correlated with difficulty completing programming tasks [1]. Generating helpful error messages has been well studied for conventional functional languages like Haskell and ML, through projects such as the Helium Haskell compiler [2, 3, 4] and counter-factual typing [5]. However, the topic has been largely unexplored for dependent types.

In this work, we take the first steps towards improving error messages in a dependently-typed language. We focus on a *higher order unification algorithm*, which is a key part of a checker for dependent types. Taking inspiration from techniques for Haskell-style languages, we provide a representation that allows for the entire unification problem-set to be considered simultaneously, allowing for heuristics to diagnose error causes and suggest hints to the programmer as to how they can be repaired. Our complete contributions include:

- An overview of problems with current dependently-typed error messages (Section 2).
- A new type of constraint graph, called a *replay graph*, which allows for global heuristic analysis of a unification problem, while accounting for the dependencies inherent to dependent typing (Section 6.2).
- A strategy for *error-tolerance*, allowing typechecking to continue even after an error is found (Section 6.4).
- *Counter-factual unification*, which, when a solution for a variable is found, proceeds by solving the unification problem both with and without the solution, allowing for multiple conflicting solutions to be regarded as equally valid (Section 7).
- The integration of these techniques into an existing unification algorithm, with proofs that they do not affect correctness (Sections 6.5 and 7.5).

*Corresponding author: Joseph Eremondi, University of British Columbia, E-mail: jeremond@cs.ubc.ca

Wouter Swierstra, Jurriaan Hage, Utrecht University, E-mail: [w.s.swierstra, j.hage}@uu.nl](mailto:{w.s.swierstra, j.hage}@uu.nl)

- An implementation of the above techniques in a simple dependently typed language, with a qualitative comparison of the generated error messages to those from existing languages (Section 8). Our implementation is able to generate repair hints and identify the true cause of the error in many cases.

While we provide a base set of heuristics as a proof of concept, the focus of this work is on developing a general framework in which heuristics can generate error messages, rather than developing the heuristics themselves.

2 Error Reporting: Principles and Status Quo

To begin, we highlight the goals of error-message generation by providing classes of unsatisfactory error messages, along with simple examples of ill-typed programs in dependently-typed languages generating such messages. We choose examples from Agda [6] and Idris [7], two (relatively) popular dependently-typed languages that heavily utilize unification. We favour these languages over ones such as Coq, since they use a relatively direct style of programming, as opposed to using tactics to automate proof generation. To keep our presentation clear, we take some liberties, abbreviating syntax and reformatting messages in minor ways. A box is used to highlight the part of the code which the message identifies as faulty.

In Section 8, we show the cases in which we are able to improve upon the error messages given by Agda and Idris, establishing that unhelpful messages are not inherent to dependent types.

2.1 Error Location and Cause

The main goal of a (type) error message is to inform the user which code they must change in order to cause the program to typecheck. Part of this involves *error-message location*, reporting one or more source-code locations as a possible cause of an error. When multiple source locations are relevant to the error, we wish to report all locations that provide the programmer with information necessary to make a repair. A simple strategy for this is to report all locations involved with an error [8]. Alternatively, heuristics can guess the ideal location, as in Helium [3, 2] and ShErrLoc [9].

Additionally, we would like error messages to indicate the *cause* of the error: the specific mistake made, whose re-

moval will cause the program to typecheck. This typically consists of a high-level, natural language message given to the programmer. Even better is to suggest a *repair*, giving not only the problem in the code, but the change that must be made to correct it. Helium generates such messages using heuristic analysis [3].

Consider the following Agda code. The function `myOp` performs arithmetic on a pair of numbers and another number. When we try to fold `myOp` over a list containing number-boolean pairs, we get a type error.

```
foldr : {A : Set} {B : Set} →
  (A → B → B) → B → List A → B
myOp : N × N → N → N
myList : List (N × Bool)

myVal : N
myVal = foldr myOp 0 myList
--Bool !<= N of type Set
--when checking that the expression myList
--has type List (N × N)
```

The reported message locates the error at `myList`, which is partially correct, since it is an argument of the wrong type. However, it is missing crucial information, namely that we are expecting a number because of the type of `myOp`. Thus, both locations are relevant to the error. This cause is hidden by the fact that this constraint is carried through the unification variables, induced by the implicit arguments of `foldr`.

Our framework allows for multiple locations be identified as the source of an error, though our proof-of-concept heuristics do not do so for this particular error. The heuristics we provide are particularly good at identifying the cause of an error and reporting a hint for how to fix it. Consider the following code snippet, where arguments to a polymorphic function are given in the wrong order.

```
myFun : (a : Set) → a → N → N
myList : List N

myApp = myFun _ 0 myList
--(List N) !<= N of type Set
--when checking that the expression myList has type N
```

Again, the user is told that the type is wrong, but not why it is wrong. Our heuristics are able to instead generate the following message, which not only reports the error, but gives its cause (arguments in the wrong order) and a possible repair (rearranging the arguments).

```
-- Mismatch in type of myFun _ (0 :: Nat) (myList)
-- Nat /= List Nat
-- HINT: Function arguments in the wrong order.
-- Try ("myFun") _ (myList) (0)
```

We give this example in more detail, along with several more examples of improvements we are able to achieve, in Section 8.

2.2 Left-to-right Bias

Depending on the order a program is traversed in a type-inference algorithm, *bias* can be introduced. Unification variables are generated for various expressions, and may be assigned a value as the algorithm progresses. If the algorithm attempts to assign another type to the same variable, that type will be marked as incorrect, even if it is the desired value and the initial value was wrong. The order in which the AST is traversed determines which type is assigned to the expression, and which type is viewed as incorrect.

This kind of bias interferes with the goals of error message generation, since both the assignment of an error location, and the reported error cause, depend on the order in which unifications are performed, as opposed to the actual location or cause.

In a bare-bones dependently-typed language, this type of bias is rare: since type inference is undecidable in general, functions must be annotated with their types, so the programmer always expresses the intended type of a function. However, when implicit arguments are introduced and values are inferred, it is once again possible for bias to arise.

Consider the following Agda code, with corresponding errors:

```
myZipWith : {A B : Set} -> ((A × A) -> B) -> List A ->
↳ List A -> List B
myVal1 =
  myZipWith proj1 (1 :: 2 :: []) (true :: false :: [])
-- Bool !=< N of type Set
-- when checking that the expression true has type N
myVal2 = myZipWith proj1 (true :: false :: []) (1 :: 2
↳ :: [])
-- N !=< Bool of type Set
-- when checking that the expression 1 has type Bool
```

The correct type is assumed to be the type of the first element of the list, even though switching the types of either list will remove the error. What’s more, the reported errors do not change if type annotations are added! We can see a similar bias in Idris:

```
myZipWith : {A B : Set} -> ((A × A) -> B) -> List A ->
↳ List A -> List B
n : Nat

myVal1 : Nat
myVal1 =
  length (myZipWith fst [n,n,n] [ True, False])
--When checking right hand side of myVal1 with
↳ expected type Nat
--When checking an application of constructor :::
--Type mismatch between Bool (Type of True) and Nat
↳ (Expected type)
myVal2 : Nat
myVal2 = length (myZipWith fst [True, False] [ n
↳ ,n,n ] )
--When checking right hand side of myVal2 with
↳ expected type Nat
--When checking an application of constructor :::
--Type mismatch between Nat (Type of n) and Bool
↳ (Expected type)
```

Various approaches exist for reducing bias. The Helium compiler [2, 3] reduces bias through constraint-based typechecking, and by representing a unification problem as a single graph, allowing for global analysis regardless of the order in which constraints are added. Counter-factual typing [5] reduces bias by also finding the solutions that would have been generated if that constraint had never been added.

Bias introduces additional complexities specific to dependently-typed languages: the ordering in which constraints are solved may affect not only which type is assumed correct, but also which value of an index for a type-family is assumed correct. Since evaluation occurs during typechecking, the first value chosen as “correct” can affect the result of evaluation, dramatically changing the types involved. A detailed example of this phenomenon, along with how our work solves it, is given in Section 7.

Our approach uses both graph-based checking (Section 6) and counter-factual typing (Section 7) to minimize bias.

2.3 Other Unsatisfactory Messages

Here we present additional examples of unsatisfactory messages. Some fall into the categories described, but use advanced type features not modeled by our core calculus, and others have messages whose deficiencies are not due to error location or bias. These problems will not be helped by the techniques we present later. However, this work is, to the authors’ knowledge, the first examining the state of dependently-typed error messages, so we catalogue them here for the sake of completeness, hoping to inspire future work.

2.3.1 Pattern Matching and Rewriting

Dependent pattern matching is a key feature in dependently-typed languages, but is also a source of complexity. With dependent pattern matching, the values matched in the left-hand side influence the expected type of the right-hand side. Additionally, matching on one argument to a function can imply specific values that the others must take.

Here we give a flawed proof of symmetry of equality. The program pattern matches the proof of $x = y$ with `refl`, its only possible value, and tries to return `refl` as a proof that $y = x$. However, `refl` returns a proof that $x = x$, causing a mismatch.

```
mysym : (A : Set) -> (x y : A) -> x ≡ y -> y ≡ x
mysym A x y refl = refl
--x != y of type A when checking that the pattern refl
↳ has type x ≡ y
```

The error informs us that $x \neq y$, but never tells us why it is expected that the variables be equal, except perhaps by indicating that `refl` is the error location. The true cause of the error is that the pattern match must also specify that x and y are equal, but the user is not informed of this. This can be repaired using Agda's dot patterns.

```
mysym A x .x refl = refl
```

The Idris equivalent faces a similar problem, and while it provides more hints as to the cause, it unhelpfully reports the function name as the location of the error.

```
mysym : (A : Type) -> (x : A) -> (y : A)
        -> x = y -> y = x
mysym A x y Refl = Refl
--When elaborating left hand side of mysym:
--When elaborating an application of Main.mysym:
--Can't unify y = y (Type of Refl)
--with x = y (Expected type)
```

The information given is similarly vague when we rewrite goals using equality proofs. The following code contains `nilNeutralR`, a proof that appending a list to the empty list does not change it, and tries to show that appending the empty list does not change the length of a list. It uses a *rewrite*, where an equality is replace simplify the goal type. However, the equality must be flipped before using it.

```
nilNeutralR : (xs : List Bool) -> xs ≡ xs ++ []
nilLengthR : (xs : List Bool) -> length xs ≡ length
↳ (xs ++ [])
```

```
nilLengthR xs rewrite nilNeutralR xs = refl
--w != w ++ [] of type List Bool when checking that
↳ the pattern refl has type w ≡ w ++ []
```

The code can be fixed by using `sym nilNeutralR` in place of `nilNeutralR`, but the error message gives no hints that this repair would solve the problem.

2.3.2 Amechanicity

Another goal of error messages is to ensure that error messages are *amechanical* [10]: we want the reported messages to avoid leaking internal details of the compiler, and to be rooted in the provided source code.

For instance, many Agda error messages will refer to intermediate metavariables generated by the unification algorithm. Consider the error for the following code, which includes metavariable names that never appear in its source code.

```
myPlus : (N × N) -> N
myVal : N
myVal = foldr myPlus 0 [ 1 ]
--N !=< (_B -> _B) when checking that the expression
↳ myPlus has type N × N -> _B -> _B
```

Uninformative messages also occur when a code snippet contains a metavariable which does not have a unique solution. Here, the messages given by Agda contain almost no clues as to the underlying cause.

```
alwaysZero = natElim [ ] (zero) (\ x y -> zero)
-- _9 : N -> Set, _10 : _9 0, _11 : _9 0, _12 : _9
↳ (suc x), _13 : _9 (suc x)
```

Idris reports a similar message, though a type mismatch is reported, as opposed to an unsolved metavariable:

```
alwaysZ : Nat -> Nat
alwaysZ = natElim [ ] (Z) (\ x, y => Z)
--When checking right hand side of alwaysZ with
↳ expected type Nat -> Nat
--When checking argument ms to function Main.natElim:
--Type mismatch between Nat (Type of 0) and m (S x)
↳ (Expected type)
--Holes: alwaysZ
```

Similarly, if unification ever gets stuck, the user is provided with an unhelpful message, referring to the internal numbers of unification problems, giving no indication of why the problems are blocked. The following example was adapted from an error encountered by an unfortunate user on Reddit [11], who was faced with the internal details of the unification solver while trying to write a simple proof.

```

postulate A : Set
postulate a : A
postulate f : A → A
postulate lem : a ≡ f a

r1 : {S : Set} {R : S → Set} {x y : S} → R x → x ≡
  ↪ y → R y
r1 R x refl = R x

K : (R : A → Set) → R a → R (f a)
K R Ra = r1 R a lem
-- Failed to solve the following constraints:
-- _23 := λ R Ra → r1 (_21 R Ra) lem [blocked on
  ↪ problem 32]
-- [32] _R_22 R (f a) =< R (f a) : Set
-- [26] (R a) =< (_R_22 R a) : Set
-- _20 := (λ R Ra → Ra) [blocked on problem 26]

```

3 Background

To describe how to generate quality error messages for dependent types, we first introduce some prerequisite concepts. We assume familiarity with the basics of type theory, dependently typed languages, and graph theory.

3.1 Preliminaries

We focus on dependently-typed languages that combine terms and types into a single syntactic category, with a type of types, denoted `Set`.

We use the notation $\overline{X_i^i}$ to represent a sequence of i elements.

We denote the operation of substituting all free occurrences of x with Y in Z as $[x \mapsto Y]Z$. Here x may be a program variable or metavariable, Y is a term, and Z may be any structure that binds these, such as terms, types, contexts, metacontexts, etc. We sometimes find it convenient to use *hereditary* substitution [12]: where evaluation to a normal form of terms is performed alongside the substitution. We denote this by $[x \overset{\rightarrow}{\mapsto} Y]Z$.

If X is a term, context, etc. then we denote the set of free program variables in X as $FV(X)$, and the set of all free and bound variables in X as $\text{vars}(X)$. The set of free metavariables in X is denoted by $\text{FMV}(X)$.

For rewrite rules, e.g., reduction \rightarrow and metacontext-solving \mapsto , we use the symbol $*$ (e.g. \rightarrow^* , \mapsto^*) to denote the reflexive and transitive closure of a relation.

3.2 Definitional Equality

Throughout this paper, we will be searching for unifiers that cause two terms to be equal. It's important to note that deciding *extensional equality*, whether two terms always behave the same way, is not possible. We instead use *definitional equality* [13], denoted $s =_{\beta\delta\eta} t$, which asserts that two terms are identical up to β -reduction, η -expansion, substitution of top-level definitions δ , and bound variable naming.

3.3 Higher Order Unification

We define the higher-order unification problem for a programming language as follows. We begin with a set $V = \{\alpha_1, \dots, \alpha_n\}$ of *metavariables*, and a set of *problems*, of the following form:

$$\forall x_1 : T_1, \dots, x_k : T_k. s : S \equiv t : T$$

where $FV(s, t, S, T, T_1, \dots, T_k) \subseteq \{x_1, \dots, x_k\}$
and $\text{FMV}(s), \text{FMV}(t) \subseteq V$

When it is clear to do so, we omit the types from our notation for problems.

The goal of higher-order unification is to find an assignment of closed terms t_1, \dots, t_n , such that for each problem:

$$\overline{\forall x_i : T_i^i. s} : S \equiv t : T$$

and for every well-typed $\overline{v_i : T_i^i}$, we have:

$$\overline{[x_i \mapsto v_i]^i [\alpha_n \mapsto t_n]^n s} =_{\beta\delta\eta} \overline{[x_i \mapsto v_i]^i [\alpha_n \mapsto t_n]^n t} : T$$

Additionally, solutions should satisfy:

$$\overline{[\alpha_n \mapsto t_n]^n S} =_{\beta\delta\eta} \overline{[\alpha_n \mapsto t_n]^n T} : \text{Set}$$

That is, we wish to find an assignment that causes both sides of each equivalence to become well typed and definitionally equal.

Higher-order unification is in general undecidable, and for many instances there is no unique solution which generalizes all other solutions. However, Miller [14] identified a sub-problem which is decidable and admits most-general unifiers: the pattern fragment, where metavariables can only be applied to distinct bound variables [15].

Unification is used in dependently typed languages for two main purposes. The first is dependent pattern matching. Here, type of the value being matched upon is unified with the constructor's return type, and the constructor's argument types are unified with the types of the matched variables. For example, when defining:

```
head : (v : Vec A (Succ n)) → A
```

the constructor `null` has return type `Vec A 0`, so when we pattern match on `v`, the case for `null` is omitted, since we cannot unify `Vec A 0` with `Vec A (Succ n)`. Conversely, the unification succeeds for the `cons` constructor. A detailed account is given by Norell [16].

In this work, however, we focus on a second use: unification of *program metavariables*. These can be used in place of expressions, when there is only one possible value the expression can have in order to typecheck correctly. In Agda and Idris, metavariables are denoted with the underscore character `_` and we will likewise use this notation in our example programs.

Program metavariables can be used in the implementation of *implicit arguments*, which allow the programmer to call a function with metavariables inserted for specific missing arguments. These are critical for hygienic polymorphism in dependently typed languages, since many use System-F style explicit instantiation of type variables. For example, the identity function has type $(T : \text{Set}) \rightarrow T \rightarrow T$, and it must be first applied to a type before it can be applied to a value, such as in `id Nat 0`. To reduce the burden on the programmer, the parameter T can be made implicit: at compile time it is replaced with a metavariable, and its value is found through unification.

3.4 Occurrences

With unification, we will often use the following terminology to describe the occurrences of terms, variables, metavariables, etc. [15]. An occurrence is *flexible* if it occurs as an argument to a metavariable, and is *rigid* otherwise. Notably, a flexible occurrence may disappear if the metavariable is given a solution that ignores its argument. We say that an occurrence is *strong-rigid* if it is not an argument to a variable or metavariable, meaning that no substitution can eliminate it.

4 The Base Language

To showcase our techniques for generating error messages, we set the stage with a dependently typed core calculus, and a conventional higher-order unification algorithm. A noteworthy aspect of our presentation is that we use a single ternary relation to capture both typechecking and finding the normal form of a term. Otherwise, the syntax and semantics are fairly standard, and serve primarily to provide the vocabulary with which we can describe our contributions in later sections.

Syntax

$s, t, S, T ::=$	x	(Program Variable)
	$t : T$	(Type Annotation)
	Set	(Type of types)
	$t \cdot \bar{e}_i^i$	(Elimination)
	$\Pi S T$	(Function type)
	$\Sigma S T$	(Pair type)
	$\lambda x. t$	(Functions)
	(t_1, t_2)	(Pairs)
$d, e ::=$	funElim $t \text{ arg}$	(Function app.)
	π_1	(Pair first proj.)
	π_2	(Pair second proj.)
$h ::=$	x	(Variable head)
	α	(Metavariable head)
$u, v, U, V ::= \dots$		
	$h \cdot \bar{e}_i^i$	(Neutral term)

$$E ::= \square \mid t : E \mid E : V \mid t E \mid E v \mid \pi_1 E \mid \pi_2 E \\ \mid \Pi E T \mid \Pi V E \mid \Sigma E T \mid \Sigma V E \mid (E, t)$$

Reduction Semantics

$$(\lambda x. s) t \rightarrow [x \mapsto t]s \\ \pi_1(s, t) \rightarrow s \\ \pi_2(s, t) \rightarrow t \\ E[t] \rightarrow E[t'] \quad \text{if } t \rightarrow t'$$

Fig. 1: Base language: syntax and semantics

4.1 Syntax

The syntax for our language is presented in Figure 1. As a dependently-typed language, types and terms inhabit a single syntactic category. We have $\Pi S T$ for functions that, when given an argument x of type S , return a value of type $T x$. The pair type $\Sigma S T$ is similar: if the first element is x of type S , the second element has type $T x$. Our language features no explicit polymorphism, but the familiar type $\forall X. T[X]$ can be simulated using $\Pi \text{Set } (\lambda X. T[X])$. We use $S \rightarrow T$ as a shorthand for $\Pi S (\lambda x. T)$ where x is not free in T .

Many presentations of dependent types use a form $\Pi (x : S) T$ or $(x : S) \rightarrow T$ for the dependent function type, but we instead represent this as $\Pi S (\lambda x. T)$, explicitly binding the argument value in a λ . These representations are conceptually the same, but ours is well suited to unification, since whenever $\Pi S T \equiv \Pi S' T'$, we can say that $S \equiv S'$ and $T \equiv T'$. Our Σ types are represented similarly.

In our syntax, elimination of terms are represented in *spine form* [17]: a sequence of eliminators is applied to a term. This gives us easy access to the head of a sequence of applications, which will prove useful in our unification algorithm. A *neutral term* is a spine term whose head is a variable or metavariable, named as such because they cannot be reduced. For readability, we will often write applications and projections using the traditional notation. For example, we write $x \cdot (\text{funElim } f, \pi_1)$ as $\pi_1(f x)$ when convenient. Similarly, we will often abuse our notation, using eliminations where a term is expected, such as on both sides of a unification problem.

The set of values is very similar to the set of terms, with the restriction that eliminators may only be applied to variables or metavariables, that is, all spine terms must be neutral terms.

In our implementation, we supported a few inductive types, such as `Nat`, `Eq`, and `Vec`, with eliminators in place of explicit recursion. While we omit these from our model for brevity (as they do not significantly affect our results), we will use them in our examples and evaluations, to provide realistic examples of dependently-typed code.

4.1.1 Semantics

The semantics of our language, shown in Figure 1, match those of a standard call-by-name lambda calculus. Function applications are evaluated through substitution, and projections extract the relevant part from a pair. Context rules allow for evaluation deep within terms. The impre-

dicative assumption that `Set : Set` means that not all terms have a normal form, but we nevertheless choose such a system for the sake of simplicity. Solutions to this problem include separating types from kinds [18], or a hierarchy of universes [19]. Such solutions are orthogonal to the work we present here.

4.1.2 Declarative Typing

In Figure 2, we define our language's type system. In the absence of metavariables, the type rules for our language are fairly standard. They are not syntax directed, but can easily be made so [20, 21].

We follow Löh, McBride, and Swierstra [20] and use a bidirectional type system, distinguishing type-synthesis judgments \Rightarrow , where types are viewed as output, from type-checking judgments \Leftarrow , where types are viewed as input. Similar to Gundry [21], we use a single relation to define both the typing of terms and their β -reduced, η -long normal forms. We write $\Gamma \vdash t \rightsquigarrow v \Leftarrow T$ to denote that a term t has normal form v and checks against type T , with a similar definition for synthesis. This allows for an easy formulation of definitional equality. It will also prove useful when defining error-tolerant typing (Section 6.4), since we can produce normal forms for terms where ill-typed subterms are replaced with \perp . The complete rules are defined in Figure 2.

The (check) and (ann) rules are standard for bidirectional systems: (check) allows us to check a term against any type with the same normal form as its synthesized type, and (ann) allows us to synthesize a type for an annotated term, provided it checks against that type. The (set) rule encodes impredicativity, and (var) simply looks up variable types in the environment.

The (pi) and (sigma) rules perform kind checking: since types and terms overlap, we must make sure that types do not refer to ill-typed terms. Function codomains and pair second-element types are encoded as λ -terms, since this will allow us to express equalities between them without worrying about the specific bound names. We ensure that these are in η -long form by substituting a fresh variable for their argument, evaluating their bodies to a normal form, and abstracting over the variable.

The (abs) and (pair) rules type introduction forms, but are checking rules rather than synthesis rules, since full inference is undecidable for dependent types. As with (pi) and (sigma), we check abstractions by substituting a fresh variable in for the argument, checking the body, and rewrapping in a λ to produce an η -long result. To check a pair construction, we check its first element, then compute

Declarative Typing

$$\boxed{\Gamma \vdash t \rightsquigarrow v \Leftarrow T} \quad t \text{ checks against type } T \text{ and has normal form } v$$

$$\boxed{\Gamma \vdash t \rightsquigarrow v \Rightarrow T} \quad t \text{ synthesizes type } T \text{ and has normal form } v$$

$$\begin{array}{c} \Gamma \vdash t \rightsquigarrow u \Rightarrow S \\ \Gamma \vdash S \rightsquigarrow U \Leftarrow \mathbf{Set} \\ \text{(check)} \frac{\Gamma \vdash T \rightsquigarrow U \Leftarrow \mathbf{Set}}{\Gamma \vdash t \rightsquigarrow u \Leftarrow T} \end{array} \quad \begin{array}{c} \Gamma \vdash S \rightsquigarrow V \Leftarrow \mathbf{Set} \\ \Gamma \vdash s \rightsquigarrow u \Leftarrow V \\ \text{(ann)} \frac{\Gamma \vdash s \rightsquigarrow u \Leftarrow V}{\Gamma \vdash (s : S) \rightsquigarrow u \Rightarrow V} \end{array} \quad \begin{array}{c} \text{(set)} \frac{}{\Gamma \vdash \mathbf{Set} \rightsquigarrow \mathbf{Set} \Rightarrow \mathbf{Set}} \end{array}$$

$$\begin{array}{c} \Gamma(x) = T \\ \text{(var)} \frac{}{\Gamma \vdash x \rightsquigarrow x \Rightarrow T} \end{array} \quad \begin{array}{c} x \notin \Gamma \quad s \ x \rightarrow^* u \\ V \ x \rightarrow^* V' \quad \Gamma, x : S \vdash u \rightsquigarrow v \Leftarrow V' \\ \text{(abs)} \frac{}{\Gamma \vdash t \rightsquigarrow (\lambda x. v) \Leftarrow \Pi U V} \end{array}$$

$$\begin{array}{c} \pi_1 t \rightarrow^* u' \quad \pi_2 t \rightarrow^* v' \\ \Gamma \vdash u' \rightsquigarrow u \Leftarrow S \quad T \ u \rightarrow^* V \\ \text{(pair)} \frac{\Gamma \vdash v' \rightsquigarrow v \Leftarrow V}{\Gamma \vdash t \rightsquigarrow (u, v) \Leftarrow \Sigma S T} \end{array} \quad \begin{array}{c} \Gamma \vdash S \rightsquigarrow U \Rightarrow \mathbf{Set} \quad x \notin \Gamma \\ \Gamma, x : U \vdash T \ x \rightsquigarrow V \Leftarrow \mathbf{Set} \\ \text{(pi)} \frac{}{\Gamma \vdash \Pi S T \rightsquigarrow \Pi U (\lambda x. V) \Rightarrow \mathbf{Set}} \end{array}$$

$$\begin{array}{c} \Gamma \vdash S \rightsquigarrow U \Leftarrow \mathbf{Set} \quad x \notin \Gamma \\ \Gamma, x : U \vdash T \ x \rightsquigarrow V \Leftarrow \mathbf{Set} \\ \text{(sigma)} \frac{}{\Gamma \vdash \Sigma S T \rightsquigarrow \Sigma U (\lambda x. V) \Rightarrow \mathbf{Set}} \end{array} \quad \begin{array}{c} \Gamma \vdash s \rightsquigarrow h \cdot \bar{e}_i^i \Rightarrow \Pi S T \\ \Gamma \vdash t \rightsquigarrow u \Leftarrow S \quad T \ u \rightarrow^* V \\ \text{(app-neutral)} \frac{}{\Gamma \vdash s \ t \rightsquigarrow h \cdot \bar{e}_i^i (\text{funElim } u) \Rightarrow V} \end{array}$$

$$\text{(app-redex)} \frac{\Gamma \vdash s \rightsquigarrow \lambda x. u \Rightarrow \Pi S T \quad \Gamma \vdash t \rightsquigarrow u' \Leftarrow S \quad T \ u' \rightarrow^* V \quad [x \mapsto u']u \rightarrow^* v}{\Gamma \vdash s \ t \rightsquigarrow v \Rightarrow V}$$

$$\begin{array}{c} \Gamma \vdash t \rightsquigarrow h \cdot \bar{e}_i^i \Rightarrow \Sigma S T \\ \text{(fst-neutral)} \frac{}{\Gamma \vdash \pi_1 t \rightsquigarrow h \cdot \bar{e}_i^i \pi_1 \Rightarrow S} \end{array} \quad \begin{array}{c} \Gamma \vdash t \rightsquigarrow h \cdot \bar{e}_i^i \Rightarrow \Sigma S T \quad T \ u \rightarrow^* V \\ \text{(snd-neutral)} \frac{}{\Gamma \vdash \pi_2 t \rightsquigarrow h \cdot \bar{e}_i^i \pi_2 \Rightarrow V} \end{array}$$

$$\begin{array}{c} \Gamma \vdash t \rightsquigarrow (u, v) \Rightarrow \Sigma S T \\ \text{(fst-redex)} \frac{}{\Gamma \vdash \pi_1 t \rightsquigarrow u \Rightarrow S} \end{array} \quad \begin{array}{c} \Gamma \vdash t \rightsquigarrow (u, v) \Rightarrow \Sigma S T \quad T \ u \rightarrow^* V \\ \text{(snd-redex)} \frac{}{\Gamma \vdash \pi_2 t \rightsquigarrow v \Rightarrow V} \end{array}$$

$$\boxed{\Gamma \vdash t : T} \quad t \text{ has type } T$$

$$\text{(hastype)} \frac{\Gamma \vdash t \rightsquigarrow u \Rightarrow T}{\Gamma \vdash t : T}$$

$$\boxed{\Gamma \vdash s =_{\beta\delta\eta} t : T} \quad s \text{ and } t \text{ are definitionally equal at type } T$$

$$\text{(defeq)} \frac{\Gamma \vdash s \rightsquigarrow u \Leftarrow T \quad \Gamma \vdash t \rightsquigarrow u \Leftarrow T}{\Gamma \vdash s =_{\beta\delta\eta} t : T}$$

Fig. 2: Declarative typing

$\Delta ::=$	\cdot	(Empty)
	$\Delta, \alpha : T$	(Declaration)
	$\Delta, ?P$	(Problem)
	$\alpha := t : T$	(Solutions)
	\perp	(Failure)
$P ::=$	$s : S \equiv t : T$	(Equality Constraints)
	$\forall \Gamma. P$	(Quantified Problems)
$\text{(meta)} \frac{\alpha : T \in \Delta}{\Gamma \Delta \vdash \alpha \rightsquigarrow \alpha \Rightarrow T}$		

Fig. 3: Metacontext syntax and typing

the expected type of the second element by substituting the value of the first into the function provided by the Σ type, then evaluating. To make pairs η -long, we evaluate the normal forms of the first and second elements and produce the pair containing them as a normal form.

The elimination rules each come in two forms: one for redexes, and one for neutral terms, although the rules differ only in the normal forms produced. To type an application, we synthesize the function type, check the argument against its expected type, and compute the return type by substituting the argument value into the codomain type and evaluating. Typing a pair's first projection is trivial. To type a second projection, we synthesize the type of the pair, then instantiate with the value of the first projection. To compute normal forms, we perform reduction in the redex case, and add the eliminator to the spine in the neutral case.

At the bottom of Figure 2, we provide two relations for convenience. One is simply type checking that discards normal forms. The second gives the meaning of definitional equality: two terms are definitionally equal at a given type if they check against that type with the same normal form.

4.1.3 Metavariables and Constraint Generation

To introduce metavariables into our language, we must first store their typing information in *metacontexts*, defined in Figure 3. These can contain declarations, where a metavariable's name and type are given, problems, which constrain two terms to be equal relative to a set of free variables, and assignments, which give concrete solutions for metavariables.

Assignments are not generated during typechecking, but will be created and used during unification.

The addition of metavariables requires that each declarative rule also take a metacontext Δ alongside the telescope Γ . However, these are always simply threaded through, except in the typing rule for metavariables. We provide this rule, along with the syntax for metacontexts, in Figure 3.

Once we add metavariables to our language, we require *global* inference in order to typecheck our programs. Both programs and types may contain metavariables that must be solved in order to complete typechecking.

To check a program with metavariables, we follow the standard approach for turning a typechecking algorithm into a constraint-based one: each time a rule pattern-matches on an input, we replace that with an equality constraint. Constraint-based typechecking take a metacontext as input, and produces as output one that contains the constraints that must be satisfied for the program to be well typed. This is then passed to a separate unification solving procedure. A full algorithm is provided by Norell [16], ensuring type-safety through *guarded constants*.

5 Higher Order Unification: A Primer

The main contribution of this work is a series of modifications to higher-order unification which facilitate the generation of quality error messages. However, to discuss modifications, we must first give the intuition behind higher-order unification. Here we present a higher-order unification algorithm, which we refer to as the *unmodified* algorithm. We highlight the intuition behind it, particularly the parts that we will modify later on. The rules we present are based on those created by Gundry and McBride [22]. We chose this algorithm for its simplicity and for its ease of implementation, as higher-order unification tends to be complex and detail-heavy. Their algorithm is in turn based on that of Abel and Pientka [15], which is somewhat cleaner theoretically, but less easily implemented.

5.1 Unification as Rewriting

Our algorithm is presented as a non-deterministic set of rules for rewriting metacontexts, which can be seen in Figure 4. We write $\Delta \mapsto \Delta'$ if we can rewrite Δ into Δ' in a single step, and use \mapsto^* to represent the symmetric-transitive closure of the rewrite relation. Details for making

$\Delta \mapsto \Delta'$		
Solving Rules		
$\Delta, \forall\Gamma. (t : T) \equiv (t' : T') \mapsto \Delta$	if $\Gamma \Delta \vdash T =_{\beta\delta\eta} T' : \mathbf{Set}$ and $\Gamma \Delta \vdash t =_{\beta\delta\eta} t' : T$	(reflexivity)
$\Delta, \forall\Gamma. (\Pi A B : \mathbf{Set}) \equiv (\Pi S T : \mathbf{Set}) \mapsto \Delta, \forall\Gamma. (A : \mathbf{Set}) \equiv (B : \mathbf{Set}) \wedge (B : A \rightarrow \mathbf{Set}) \equiv (T : S \rightarrow \mathbf{Set})$		(rigid-pi)
$\Delta, \forall\Gamma. (\Sigma A B : \mathbf{Set}) \equiv (\Sigma S T : \mathbf{Set}) \mapsto \Delta, \forall\Gamma. (A : \mathbf{Set}) \equiv (B : \mathbf{Set}) \wedge (B : A \rightarrow \mathbf{Set}) \equiv (T : S \rightarrow \mathbf{Set})$		(rigid-sigma)
$\Delta, \forall\Gamma. x\bar{d}_i^i \equiv x\bar{e}_i^i \mapsto \Delta, \forall\Gamma. \bar{d}_i \equiv \bar{e}_i$		(rigid-spine)
$\Delta, \alpha : T, \Delta', \forall\Gamma. \alpha\bar{x}_i^i \equiv t \mapsto \Delta, \Delta', \alpha := \lambda\bar{x}_i^i. t : T$		(inversion)
	if \bar{x}_i^i is linear on $\mathbf{FV}(t)$, $\alpha \notin \mathbf{FMV}(\Delta', t)$ and $\cdot \Delta, \Delta' \vdash \lambda\bar{x}_i^i. t : T$	
$\Delta, \alpha : \Pi\Phi. T, \Delta', \forall\Gamma. \alpha\bar{x}_i^i \equiv \alpha\bar{y}_i^i \mapsto \Delta, \Delta', \beta : \Pi\Phi. T, \alpha := \lambda\Phi. \beta\Psi : \Pi\Phi. T$		(intersection)
	if \bar{x}_i^i agrees with \bar{y}_i^i on $\Psi \subset \Phi$ and $\mathbf{FV}(T) \subset \mathbf{vars}(\Psi)$	
Failure Rules		
$\Delta, \forall\Gamma. (s : T) \equiv (t : T) \mapsto \perp$	if s and t have different rigid heads	(rigid-fail)
$\Delta, \alpha : T, \Delta', \forall\Gamma. \alpha\bar{x}_i^i \equiv t \mapsto \perp$	if α occurs rigid-strong in t	(occurs-check)
$\Delta, \forall\Gamma. \alpha\bar{d}_i^i \equiv t \mapsto \perp$	if $\mathbf{FV}^{\text{rigid}}(t) \not\subset \mathbf{FV}(\bar{d}_i^i)$	(prune-fail)

Fig. 4: Higher order unification, adapted from Gundry and McBride [22]

the rewrite rules syntax directed, as well as a sample implementation, are given by Gundry and McBride [22].

The algorithm we present is *dynamic*: when it encounters problems outside of the pattern fragment, it simply defers their solutions. Since other problems may refer to the same metavariables, substituting in their solutions may move other problems into the pattern fragment. Because of this dynamic property, our algorithm may become *stuck* if no failure has been encountered, but no remaining problems are in the pattern fragment. We use the following terminology to refer to the different states a metacontext can take.

Definition 1. *We say that a metacontext Δ is solved if it contains no problems. A metacontext is stuck if it is not solved, but no progress can be made. A metacontext is said to be final if it is solved, stuck, or \perp .*

Our rules can be grouped into three broad categories:

- **Solution Rules:** When there is only one possible solution t' to a problem $\forall\Gamma. s : T \equiv t : T$ containing a metavariable α , then we rewrite it to be $\alpha := t' : T$. Similarly, reflexive equations can be deleted.
- **Failure Rules:** These rules identify problems that have no solution, or that cannot possibly be moved into the pattern fragment.

- **Simplification Rules:** These rules are used to manipulate individual problems or parts of the metacontext, progressing towards a form that can be handled by the other rules. For example, metacontexts can be permuted in dependency-respecting ways. Likewise, to propagate our solutions, when $\alpha := t : T$ occurs in a metacontext, we replace α with t everywhere else. Other rules perform type-directed manipulation of context. For instance, problems comparing functions f, g of Π -types use η -expansion to instead compare the function bodies.

In Figure 4 we define the solution and failure rules, and explain them in the following sections. Simplification rules are critical for applying unification in a practical setting, since they greatly increase the number of problems that can be solved. However, their interactions with our modifications are minimal, so we omit these rules.

5.1.1 Solution Rules

Rigid Equations

We have a *rigid-rigid* matching when unifying two terms, where neither is an elimination with metavariable head.

To match two expressions with the same variable head, we can generate equalities between their spines, as seen in

the (rigid-spine) rule. Similarly, matching two types with the Σ or Π constructors generates matches between their arguments. For example, $\Pi \alpha T \equiv \Pi \beta T'$ is decomposed into $\alpha \equiv \beta, T \equiv T'$.

We abuse our notation slightly to match eliminators as well as terms, performing similar decomposition with the eliminator constructors as with term constructors.

Inversion and Intersection

A problem is in the pattern fragment if any elimination with a metavariable head is applied to a spine of distinct program variables. We make progress when one or both sides of an equation are an elimination of this form, since these equations have unique solutions.

A *flex-rigid* equation is one where one side of the equation is a spine-form term with a metavariable head. (Note that the spine may be empty, for example, with constraints of the form $\alpha \equiv t$). Similarly, a *flex-flex* equation is one where both sides of the equation are spine-form terms with a metavariable head.

To solve a flex-rigid equation $\alpha \overline{x_n}^n \equiv t$, if $\overline{x_n}^n$ is a sequence of distinct bound variables (i.e. the problem is in the pattern fragment), we generate a solution $\alpha := \lambda \overline{x_n}^n. t$. This process is known as *inversion*, and can be seen in the (inversion) rule. Flex-flex equations with different metavariable heads can be solved in the same way. If the problem is not in the pattern fragment, we wait, hoping that application of other rules will simplify it to be in the pattern fragment.

A special case, however, occurs when our problem is $\alpha \cdot \overline{x_i}^i \equiv \alpha \cdot \overline{y_i}^i$, that is, both equations have the same metavariable head. Here we instead solve by intersection: a fresh metavariable β is declared, and we generate the solution $\alpha := \lambda x_1 \dots x_n. \beta z_1 \dots z_k$, where $\{z_1, \dots, z_k\} = \{x_i \mid x_i = y_i\}$. That is, the solution to such a problem can only refer to the variables shared by both sides of the equation: α is defined as a function which ignores all arguments on which x_i and y_i disagree. This process is formalized in the (intersection) rule.

5.1.2 Failure Rules

Rigid-Rigid mismatch

All expressions, other than spine forms with metavariable heads, have *rigid heads*. The rigid head can be a program variable at the head of a spine, or the Π, Σ or **Set** type constructors. If two expressions have different rigid heads, since our language has injective constructors, they cannot be unified and we fail with the (rigid-fail) rule. Note that

injectivity may not apply in univalent settings, where not all constructors are injective [23].

Occurs-check Failure

When solving for a flex-rigid or flex-flex equation $\alpha \overline{e_i}^i \equiv t$, we must ensure there are no places where α occurs rigid-strong in t . If such an occurrence exists, then there is no finite term satisfying the equation, and we fail with the (occurs-check) rule. We only fail for rigid strong occurrences since other occurrences may disappear when solutions for other variables are found.

For example, there is no finite solution to $\alpha \equiv \Pi \alpha \beta$, and our occurs check would fail in this case.

Pruning

Before solving a flex-rigid or flex-flex equation, we *prune* the equations. Similar to the occurs check, when solving $\forall \Gamma. \alpha \cdot \overline{e_i}^i \equiv t$, we need to ensure all variables in t (which are bound by the $\forall \Gamma$) also occur in the spine $\overline{e_i}^i$, since only those variables can be incorporated into a solution for α .

If any are not in the spine, but are arguments to a metavariable within t , we can wait, since future solutions may make these disappear. In any other case, there can be no solution, so we fail with the (prune-fail) rule. For example, $\forall x, y. \alpha x \equiv y$ has no closed solution for α , since α may appear in contexts where y is not bound.

We present only the rules for when pruning fails, since the rules where it succeeds are simplification rules. A complete account of pruning is given by Abel and Pientka [15].

Type Checks

After a substitution has been applied, a problem $\forall \Gamma. s : S \equiv t : T$ may be solved, that is, $s = t$, and the (reflexivity) rule can be applied. However, to consider the problem solved, S and T must also be equal, and the solution must typecheck against T . If this succeeds, the problem can be removed from the metacontext. If this fails, then the problem is stuck. Note that the typechecking here occurs relative to the declarative system: we do not generate further constraints when typechecking here. Likewise, if a problem has no metavariables but its two sides are not equal, the metacontext has no solution.

Our rewrite rules use these type checks as guards when adding solutions or deleting reflexive equations, so it becomes stuck when failures are encountered. In Section 6.4 we detail how to proceed despite these failures.

6 Replay Graphs for Message Generation

Here, we describe one approach for dependently-typed error message generation. Specifically, we follow the approach of the Helium Haskell Compiler [2, 3, 4], where a unification problem is represented as a graph, with edges representing equality. This format naturally captures the transitive nature of equality using graph reachability. More importantly, it represents the entire unification problem in a global, non-linear way. Thus, instead of iteratively solving a problem and reporting an error as soon as it is encountered, a set of heuristics can be applied to diagnose the problem. These heuristics can see the entire graph, and thus the context surrounding an error. Additionally, they can try modifications on the graph, seeing which make it consistent, using this information to generate repair hints.

Many heuristics of Helium can be adapted to dependent types with little modification. However, Helium’s graph representation is too simple for dependent types, and does not naturally capture the dynamic aspects of higher-order unification.

In this section, we present our solution to this problem. We give a modified unification algorithm that generates a *replay graph*, representing the steps unification took. Instead of generating solutions from the graph, we perform unification as we normally would, but if an error is encountered, we refer to the graph, applying heuristics to generate a message.

Additionally, since replay graphs capture any failures during unification, they allow us to make our algorithm error-tolerant. When a failure is encountered, we can safely proceed as if it had not happened, since the failure is recorded in the replay graph. We formalize this process in Section 6.4.

6.1 Constraint Graphs for First-Order Unification

We begin by explaining first-order unification, since our higher-order replay graphs are easiest to understand in relation to the constraint graphs that systems such as Helium [3, 4] use for unification.

First, consider a set of equality constraints over metavariables and atomic types. To represent such a program graphically, we have nodes in our graph for each atomic type or metavariable. For each constraint $T_1 \equiv T_2$, we add an undirected edge between the nodes for T_1 and T_2 . Since reachability and equality are both transitive, we can

identify the solution for a metavariable as the atomic type in its connected component. If two distinct atomic types are in the same connected component, then no solution to the constraint set exists.

To account for types involving type constructors, such as \rightarrow or **List**, we use *directed structure edges*. First, we must establish how to construct a node for a complex type T . If T is $\chi S_1 \dots S_n$ for some type constructor χ and types $S_1 \dots S_n$, then we add a *constructor application node* to our graph for T , an atomic *constructor node* for χ , and the nodes for each S_i . The node for each S_i is either its atomic type or metavariable node, or its constructor application node if it is a complex type. To reflect the structure of T , we add a directed edge from T to χ labeled $(\downarrow, \text{HEAD})$, and a directed edge from χ to T labeled (\uparrow, HEAD) . Similarly, we add an edge (\downarrow, i_χ) from T to S_i and (\uparrow, i_χ) from S_i to T .

For instance, the type $T = T_1 \rightarrow T_2$ would be represented as a node, with the following edges:

$$(T, \rightarrow, \downarrow, \text{HEAD}), (\rightarrow, T, \uparrow, \text{HEAD}), \\ (T, T_1, \downarrow, 1\rightarrow), (T_1, T, \uparrow, 1\rightarrow), (T, T_2, \downarrow, 2\rightarrow), (T_2, T, \uparrow, 2\rightarrow)$$

As an intuition behind this notation, we imagine the syntax tree for a term with the root at the top. The edges labeled \downarrow represent moves down, from the root towards the leaves, whereas edges labeled \uparrow are moves from the leaves towards the root.

With a notion of how to represent complex terms as graphs, the final detail is how to represent equalities between complex nodes. We can form paths between nodes using both directed and undirected edges, but this includes too many paths: if we have $S_1 \rightarrow T_1$ connected to $S_2 \rightarrow T_2$, we do not want a path between S_1 and T_2 , since that equality is not implied.

We define an *equality path* inductively as follows:

- Any undirected path is an equality path.
- A path is also an equality path if it has the form

$$\overline{U}_i^i, (S, T, \uparrow, L), P, (T, S, \downarrow, L), \overline{U}_j^j$$

where P is a smaller equality path, \overline{U}_i^i and \overline{U}_j^j are undirected paths, S and T are nodes, and $L \in \{\text{HEAD}, 1_\chi, 2_\chi, \dots\}$.

Essentially, moves along directed edges must be well nested, with \downarrow preceding \uparrow . This corresponds to our equality rules for constructors: if two terms are equal, then they must have equal constructors and equal arguments. If an equality path connects two non-equal nodes for atomic types or type constructors, then it is called an *error-path*. Labelling the argument nodes with the constructor χ ensures that we do

not have redundant equality paths between arguments of terms with different heads, since there is already an error path between the heads.

An alternate, but equivalent, way to think of an equality path is by *implicit edges*. Whenever there is a path between two complex terms, there are implicit undirected edges between their heads and corresponding arguments.

A set of unification constraints has a solution if and only if two conditions hold. First, the graph for the constraint set must not contain any error paths. Secondly, no metavariable can have a path to itself through a directed edge, since this prohibits finite solutions. The second condition is essentially an occurs-check.

The advantage of this approach is that the graph provides a single representation of the entire unification problem. The various equalities can be considered simultaneously by heuristics, diagnosing the type error by selecting a set of undirected edges whose removal will eliminate all error paths. The program points that induced the constraints for these edges are then presented as the cause of the type error. Edges can be added or removed, so that heuristics can search for specific changes that will repair the graph. Moreover, the representation is unbiased: the order in which constraints are emitted has no effect on the final graph.

6.2 Replay Graphs for Higher-Order Unification

In this section, we introduce a graph representation of higher-order unification problem sets, which we call *replay graphs*. We describe methods for generating them from constraints, as well as for analyzing them in order to diagnose the likely cause of errors.

The first-order approach of Section 6.1 fails in a higher order setting. Since we are performing unification on both types and terms, we may have to unify terms involving applications of non-injective functions to types and type indices. The equality paths of first-order unification are correct only because of the injectivity of constructors. For example, the following equation holds:

$$(\lambda x. \lambda y. \mathbf{Zero}) \mathbf{Zero} \mathbf{Zero} \equiv (\lambda x. \lambda y. x) \mathbf{Zero} (\mathbf{Succ} \mathbf{Zero})$$

However, we cannot conclude from this that $(\lambda x. \lambda y. \mathbf{Zero}) \equiv (\lambda x. \lambda y. x)$, or that $\mathbf{Succ} \mathbf{Zero} \equiv \mathbf{Zero}$.

Likewise, the simplification rules of our unification algorithm transform problems so that solutions can be found. For example, η -expansion transforms unification of functions into unification of their bodies, and decomposes constraints on dependent pairs to constraints on their com-

ponents. Similarly, when a solution for a metavariable is found, the new value is substituted into terms of other problems, which are then evaluated. Such operations do not have an obvious graph-based counterpart.

In order to account for this, we decompose our problem into a number of first-order equalities by running higher-order unification and recording the sub-problems and solutions that are generated. When an error is found and a message needs to be generated, the replay graph is analyzed to diagnose the cause of the error. Thus, the graph can be ignored when finding solutions for metavariables, but it can be a rich source of information about unifications that have taken place whenever an error message needs to be generated.

Because replay graphs rely on our unmodified unification algorithm, they can exhibit some bias, as the order in which unification problems are solved can still somewhat affect the error message output. Unlike with first order problems, we cannot solve the problems in any order, since problems outside of the pattern fragment can never be solved until other solutions push them into the pattern fragment. Such restrictions are inherent to dynamic unification with dependent types, but we use counter-factual unification (Section 7) to explore all orderings which do not cause unification to become stuck.

6.2.1 Graph Structure

Our graph structure is very similar to the first order version, though enriched to allow for a unified type-term language. At its core, our graph consists of a set of nodes, which represent terms, and a set of edges between those nodes, representing structural dependencies and equalities between nodes. The specification for nodes and edges is given in Figure 5(a).

Nodes

Our nodes come in four varieties. There is a single node for each metavariable. Nodes for *rigid heads* are also present, representing the injective constructors of our language, such as \mathbf{Set} , Π or Σ . Constructor application nodes represent simple terms built from constructors, like $\Pi S T$.

To account for the remaining, more complex terms, such as lambdas or neutral terms, we have *raw term nodes*, which simply store an entire term in a single node. These act as placeholders, storing the term so we can perform substitution and evaluation on it as we find solutions for metavariables. They then provide a connection between a term and its evaluation.

Vertices		
$\mathcal{V} ::=$	α	(Metavariable Node)
	r	(Rigid Head)
	App_t	(Rigid Head Application)
	$\text{Raw}(t)$	(Raw Term node)
Rigid Heads		
	$r ::= \text{Set}[\Pi \Sigma \text{funElim}] \pi_1 \mid \pi_2 \mid x$	
Edges		
$D ::=$	$\uparrow \mid \downarrow$	
$L ::=$	$\text{HEAD} \mid i_\chi \quad i \in \mathbb{N}$	
$\mathcal{E} ::=$	$(\mathcal{V}_1, \mathcal{V}_2, D, L)$	(Applied term)
	$[\mathcal{V}_1, \mathcal{V}_2]$	(Equality Edge)

$\mathcal{V}(t) = \mathcal{V}$	Term Vertices
	$\mathcal{V}(t) = t \quad \text{if } t \text{ is a rigid head}$
	$\mathcal{V}(t) = \mathcal{V}(t') \quad \text{if } t \rightarrow t'$
	$\mathcal{V}(\alpha) = \alpha$
	$\mathcal{V}(\Pi S T) = \text{App}_{\Pi S T}$
	$\mathcal{V}(\Sigma S T) = \text{App}_{\Sigma S T}$
	$\mathcal{V}(\text{funElim } t) = \text{App}_{\text{funElim } t}$
	$\mathcal{V}(\alpha \bar{e}_i^i) = \text{Raw}(\alpha \bar{e}_i^i)$
	$\mathcal{V}(\lambda x. t) = \text{Raw}(\lambda x. t)$
	$\mathcal{V}((s, t)) = \text{Raw}((s, t))$

(b) Term Vertices

(a) Replay Graphs: Syntax

$\mathcal{E}(t) = \bar{\mathcal{E}}_i^i$	Term Edges
$\mathcal{E}(t) =$	$\cdot \quad \text{if } t \text{ is a rigid head or metavariable}$
$\mathcal{E}(\Pi S T) =$	$(\text{App}_{\Pi S T}, \Pi, \downarrow, \text{HEAD}), (\text{App}_{\Pi S T}, S, \downarrow, 1), (\text{App}_{\Pi S T}, T, \downarrow, 2)$ $(\Pi, \text{App}_{\Pi S T}, \uparrow, \text{HEAD}), (S, \text{App}_{\Pi S T}, \uparrow, 1), (T, \text{App}_{\Pi S T}, \uparrow, 2)$
$\mathcal{E}(\Sigma S T) =$	$(\text{App}_{\Sigma S T}, \Sigma, \downarrow, \text{HEAD}), (\text{App}_{\Sigma S T}, S, \downarrow, 1), (\text{App}_{\Sigma S T}, T, \downarrow, 2)$ $(\Sigma, \text{App}_{\Sigma S T}, \uparrow, \text{HEAD}), (S, \text{App}_{\Sigma S T}, \uparrow, 1), (T, \text{App}_{\Sigma S T}, \uparrow, 2)$
$\mathcal{E}(\text{funElim } t) =$	$(\text{App}_{\text{funElim } t}, \text{funElim}, L), (\text{App}_{\text{funElim } t}, \mathcal{V}(t), R)$
$\mathcal{E}_{\mathcal{G}}(\Delta) = \bar{\mathcal{E}}_i^i$	Metacontext Edges
$\mathcal{E}_{\mathcal{G}}(\cdot) =$	\cdot
$\mathcal{E}_{\mathcal{G}}(\Delta, \alpha : T) =$	$\mathcal{E}_{\mathcal{G}}(\Delta)$
$\mathcal{E}_{\mathcal{G}}(\Delta, \alpha := t : T) =$	$\mathcal{E}_{\mathcal{G}}(\Delta), \mathcal{E}(\text{subTms}(t)), [\mathcal{V}(\alpha), \mathcal{V}(t)]$
$\mathcal{E}_{\mathcal{G}}(\Delta, (s : S) \equiv (t : T)) =$	$\mathcal{E}_{\mathcal{G}}(\Delta), \mathcal{E}(\text{subTms}(s, t)), [\mathcal{V}(s), \mathcal{V}(t)]$
$\mathcal{E}_{\mathcal{G}}(\Delta, \forall \Gamma. P) =$	$\mathcal{E}_{\mathcal{G}}(\Delta), \mathcal{E}_{\mathcal{G}}([\Gamma \Rightarrow \Gamma'] P) \quad \text{where } \Gamma' \cap \text{FV}(\mathcal{G}) = \emptyset$
$\mathcal{G}_{\text{eval}}(\mathcal{G}, \Delta) = \mathcal{G}'$	Evaluation Edges
$\mathcal{G}_{\text{eval}}(\mathcal{G}, \Delta) =$	$\mathcal{G} \cup \{[\mathcal{V}(s), \mathcal{V}([\alpha \Rightarrow t]s)] \mid \alpha := t : T \in \Delta, \text{Raw}(s) \in \mathcal{G}, \alpha \in \text{FMV}(s)\}$

(c) Graph construction rules: edges

Fig. 5: Replay Graph Construction

In our implementation, we allow for multiple nodes representing the same term to be in our vertex set. This ensures that unrelated program segments do not conflate their errors. For instance, if there are two unrelated errors involving the `Set` constructor, having a path between them through a single `Set` constructor could result in a diagnosis completely unrelated to the actual cause of the error.

Edges

Our edges come in the same two varieties as first-order graphs. Undirected equality edges are written $[u, v]$. These denote that the terms defined by two nodes should be definitionally equal. Directed structure edges are written (u, v, d, L) , where $D \in \{\uparrow, \downarrow\}$, and L is either `HEAD` or i_χ for some constructor χ , denoting the children corresponding to the rigid head and i th arguments applied to constructor χ . These are used to build terms from constructors. For example, the node for $\Pi S T$ is an application node, with \downarrow edges to Π, S and T , labeled `HEAD`, 1_Π , 2_Π respectively, and corresponding \uparrow edges in the opposite direction.

6.2.2 Constructing Replay Graphs

As is standard, a graph \mathcal{G} is simply a pair consisting of a set of vertices, and a set of edges over those vertices. The graph union $\mathcal{G}_1 \cup \mathcal{G}_2$ is simply the union of its vertex and edge sets.

Overall, we wish to represent each equality encountered during unification in our graph. We add equality edges in three cases: between the two terms of a unification problem, between a metavariable and its solution, and between a term and its evaluated form after applying previous solutions.

The formal rules for constructing our graph are given in Figure 5. We gradually build the graphs for a unification run. First, we have the vertex $\mathcal{V}(t)$ for each subterm t , given in Figure 5(b). Raw terms, rigid heads, and metavariables have their own nodes, while terms with constructor heads are given application nodes. Note that we abuse our syntax slightly, also assigning nodes to eliminators.

The rules for generating edges are given in Figure 5(c). To build the complete graph for term t , we compute its vertex, and add the edges, denoted $\mathcal{E}(t)$, that represent the structure for the term. This set is empty for metavariables, rigid heads and raw terms, but contains the `HEAD` and numbered argument edges for the complex term nodes described in Section 6.1.

The graph for a metacontext Δ contains $\mathcal{V}(t)$ and $\mathcal{E}(t)$ for each sub-term t of Δ . Additionally, it contains undi-

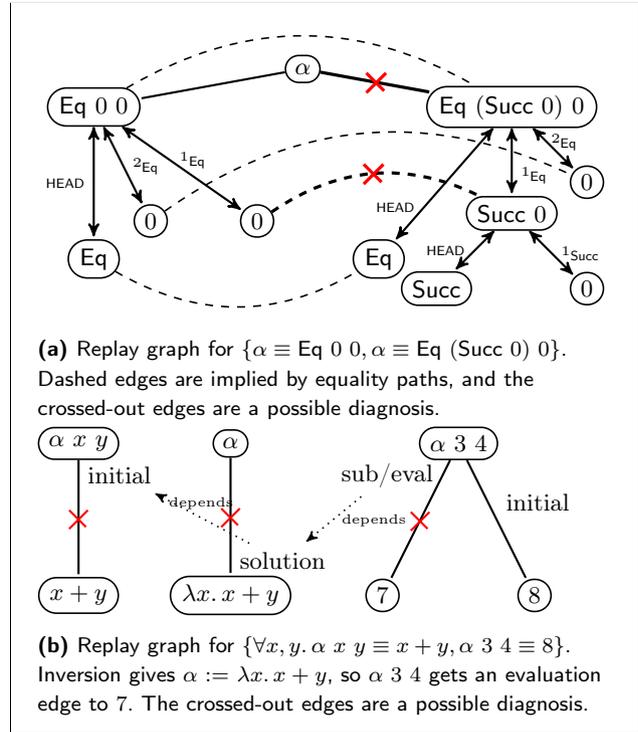


Fig. 6: Example replay graphs

rected equality edges for each equality and solution assignment contained in Δ . We denote this graph as $\mathcal{E}_{\mathcal{G}}(\Delta)$, referencing the graph \mathcal{G} constructed so far so that we can generate fresh variables.

When adding a constraint of the form $\forall \Gamma. C$ to our graph, we replace each variable in Γ with a free, unconstrained variable, ensuring that the constraints are indeed satisfied for any possible Γ .

Finally, we have rules to construct the *evaluation edges* for a metacontext Δ . These account for the fact that dependent type checking must perform evaluation of terms. When we have a solution $\alpha := t$ in Δ , and α occurs in some term s in Δ , we add an edge from s in its original raw form to s after substituting t for α and evaluating. These rules are crucial for adapting type graphs to dependent types, and account for Helium’s lack of support for type-level computation.

The connected components (via equality paths) of our graph are called *equivalence groups*: any two terms in an equivalence group should be equal after the appropriate substitutions for metavariables are made.

Example: Metacontext Graphs

Figure 6(a) shows the graph for the metacontext:

$$\Delta = \alpha : \text{Set}, ?\alpha \equiv \text{Eq } 0\ 0, ?\alpha \equiv \text{Eq } (\text{Succ } 0)\ 0$$

(While the inductive types Eq and constructors 0 , Succ are not part of our calculus, we use them to provide more realistic examples.) The nodes consist of rigid heads for 0 , Succ , and Eq , a metavariable node α , and constructor application nodes for $\text{Succ } 0$, $\text{Eq } 0 \ 0$ and $\text{Eq } (\text{Succ } 0) \ 0$.

For term edges, we have directed edges from $\text{Eq } 0 \ 0$ to nodes for Eq , 0 and 0 , labeled HEAD , 1_{Eq} and 2_{Eq} respectively, with similar edges in place for $\text{Eq } (\text{Succ } 0) \ 0$ and $\text{Succ } 0$.

Finally, we have the equality edges between α and the two Eq terms. These edges, combined with the directed edges, induce the following equality paths:

$$\begin{array}{l} \text{Eq}, (\uparrow, \text{HEAD}), \text{Eq } 0 \ 0, \quad \alpha, \quad \text{Eq } (\text{Succ } 0) \ 0, (\downarrow, \text{HEAD}), \text{Eq} \\ 0, (\uparrow, 1_{\text{Eq}}), \text{Eq } 0 \ 0, \quad \alpha, \quad \text{Eq } (\text{Succ } 0) \ 0, (\downarrow, 1_{\text{Eq}}), \text{Succ } 0 \\ 0, (\uparrow, 2_{\text{Eq}}), \text{Eq } 0 \ 0, \quad \alpha, \quad \text{Eq } (\text{Succ } 0) \ 0, (\downarrow, 2_{\text{Eq}}), 0 \end{array}$$

There is an error path between 0 and $\text{Succ } 0$. One possible diagnosis, denoted by the crossed out edges, is to delete the edge between α and $\text{Eq } (\text{Succ } 0) \ 0$, thus deleting the equality path between 0 and $\text{Succ } 0$. However, deleting the edge between $\text{Eq } 0 \ 0$ and α is an equally valid diagnosis, and heuristics must be used to determine the one that is the likely error cause.

Constructing the Complete Graph

Given the rules for constructing a graph for a metacontext, we can construct the replay graph for an entire unification run.

Suppose the unmodified unification algorithm, when given initial metacontext Δ_1 , rewrites it as the sequence $\Delta_1 \mapsto \Delta_2 \mapsto \dots \mapsto \Delta_n$, where Δ_n is final. Then we can define a series of graphs for each step as follows:

$$\begin{aligned} \mathcal{G}_0 &= \emptyset, \\ \mathcal{G}_i &= \mathcal{G}_{\text{eval}}(\mathcal{G}_{i-1} \cup \mathcal{E}(\mathcal{G}_{i-1}, \Delta_i)) \end{aligned}$$

That is, we build our final graph adding the edges from each intermediate metacontext Δ_i , then adding the edges between terms and their evaluated forms after applying solutions from Δ_i .

We say that \mathcal{G}_n is the *replay graph* of unification of Δ_1 . The graph begins empty, and then has nodes and structure edges for all terms in Δ_1 added. Each time a new solution is generated or a problem is decomposed by the unification algorithm, the corresponding edges are generated, and will be included in the final graph. This provides us with a global overview of the constraints induced by the initial metacontext, allowing us to use heuristics to diagnose errors and recommend repairs.

6.2.3 Edge Dependencies

In a practical implementation, to facilitate message generation, we can keep a dependency tree of generated constraints. The sub-constraints of a rigid-rigid unification or η -expansion P are dependent on P , and the definition $\alpha := t$ from solving a flex-rigid or flex-flex problem P is dependent on P . Updates are dependent on their definitions (which in turn may be dependent on other problems). Thus, when an edge \mathcal{E} is diagnosed as the source of the error, we can locate the error at the original source-code that induced the constraint.

Example: Evaluation Edges

Figure 6(b) shows a replay graph (with directed edges omitted) for the following metacontext Δ_1 :

$$\alpha : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}, \forall x, y. \alpha \ x \ y \equiv x + y, \alpha \ 3 \ 4 \equiv 8$$

Since $\alpha \ 3 \ 4 \equiv 8$ is not in the pattern fragment, unification begins by applying inversion on the first problem, yielding $\alpha := \lambda x \ y. x + y$. We can then substitute this for α to obtain the problem $3 + 4 = 8$, which fails.

In our replay graph, the metacontext graph for Δ_1 contains an equality edge between the raw terms $\alpha \ x \ y$ and $x + y$, and an equality edge between the raw term $\alpha \ 3 \ 4$ and the rigid head 8 . The solution from inversion creates a new metacontext with the edge between the metavariable node α and the raw term $\lambda x \ y. x + y$. We then use $\mathcal{G}_{\text{eval}}$ to construct the evaluation edge from the raw term $\alpha \ 3 \ 4$ to the rigid head 7 .

This example shows how the raw term nodes act as connectors in the graph, connecting the definition from the initial problem set to the results of evaluation.

6.3 Error Diagnosis and Heuristics

When the unification solving is complete, we have a final replay graph G . A constraint set is unsatisfiable if two incompatible nodes are connected by an equality path: a sequence of edges where moves up and down directed edges are properly nested (as defined in Section 6.1). A pair of nodes are incompatible if they are distinct rigid heads, or if one is a rigid head and one is a constructor application. Such a path is called an error path.

A *diagnosis* for an error path is a set of edges which, when removed, also deletes the error path. In order to deal with the density of our graph, we implicitly remove all edges from constraints dependent on C when we remove C . This prevents error messages from containing information about intermediate constraints which are not directly

found in the initial metacontext, and thus have a nebulous source location. Figure 6 shows two replay graphs, along with a possible diagnosis for each.

There are, of course, many possible diagnoses for an unsatisfiable path. Knowing the correct diagnosis is not always possible, since in many cases this requires knowing the programmer’s intention. As in Helium, we use two sorts of heuristics for deciding which edges to blame. *Avoid* heuristics mark certain edges unfit for removal. *Voting* heuristics, conversely, assign votes to each edge. The edge which, after running all heuristics, has the most votes and is not marked unfit, will be removed. This process is repeated until the two conflicting nodes are disconnected.

The heuristics we discuss here form a solid base for diagnosis, but further research is necessary to develop more refined heuristics.

Edge Information and Inserting Messages

Each edge, when generated, is paired with information about its creation, location in the source code, etc. This information can be accessed by the heuristics, allowing them to use this information when diagnosing the probable cause of the error. The edge information is also accessed during error message generation, so that the printed messages can be as detailed as necessary.

When a heuristic is examining an edge, it can also modify the stored information for that edge. Through this mechanism, we can suggest fixes or provide other information which was obtained during the global analysis of the replay graph, enriching the textual errors presented to the user.

First Order Heuristics

Not all of the Helium heuristics are applicable to dependent types. We outline here the heuristics from Helium [3], along with some variants, which we included in our implementation.

Participation-ratio: Edges on multiple error paths are more likely to be the cause of an error, so we avoid those on few paths.

First-come first-blamed: When all else is equal, ties are broken arbitrarily.

Permutation: If swapping two arguments to a function can repair the replay graph, the application edge is blamed and the swap is recommended as a fix.

Application: The expected number of arguments for a function is determined from a replay graph. Too many are given if an argument is given to a value without a Π type, and too few when a value is used where a non- Π type

is expected. Adding or removing arguments is suggested as a possible fix.

Dependently-typed Heuristics

Some of the core Helium heuristics can be adapted to the classes of errors which are specific to dependently-typed languages.

One key difference from simple types is the presence of *type indices*. While our core calculus has no inductive types, most languages (including the one used in our implementation) contain types which are indexed by values, such as $\text{Eq } x \ y$, the type proofs that x and y are equal.

The Helium heuristics for permuting function arguments and tuple elements can be adapted to permute type indices: if permuting the indices of an indexed type removes the error paths from the replay graph, we suggest a fix rearranging them to the user.

6.4 Error-Tolerant Typing

We now describe modifications we made in order to make higher-order unification *error-tolerant*: that is, able to continue with unification even after an error has been encountered and it is known there is no solution for a certain metavariable. Sometimes, the repair to a type error may modify or delete code causing other errors, so we do not want the programmer to only see the first error encountered, as it may be a symptom of a larger problem.

The core idea is to introduce an explicit error value into our language. Thus, when a metavariable has no solution, we simply assign it an error value and proceed normally.

The modifications to our unification algorithm are described below, with the full details in Figure 7. The error-tolerant $\hat{\mapsto}$ rewrite relation is assumed to be identical to \mapsto except for the changes we highlight. Notably, we never have $\Delta \hat{\mapsto} \perp$, allowing us to proceed even after errors are discovered.

6.4.1 Rigid-rigid Mismatches

Any time a rigid-rigid conflict is encountered (i.e. mismatched constructors), unmodified unification throws an error. Instead, we simply take no action, generating no further subproblems from the given equation.

Because rigid-rigid expansion is defined in the exact same way as derived edges in our replay graph, unmodified unification encounters a rigid-rigid error if and only if two disjoint constructors are connected in our final replay

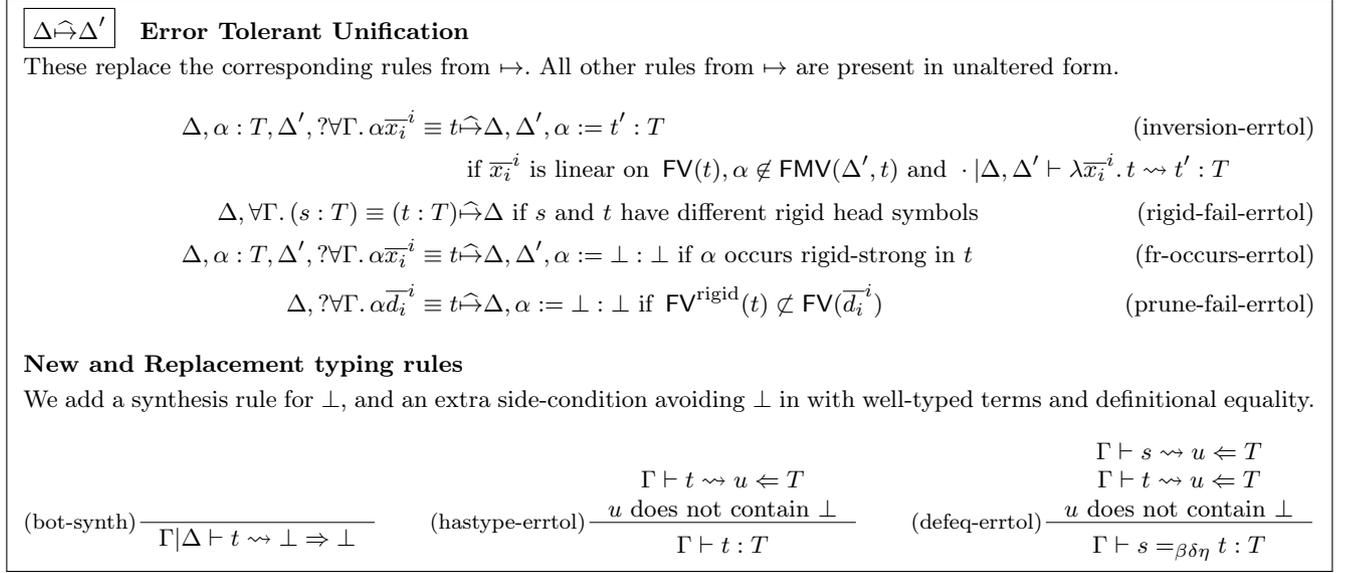


Fig. 7: Modified error-tolerant unification, typing and normal forms

graph. We can continue as if the constraint were solved, knowing that the error is reflected in the graph and will be detected.

6.4.2 An Explicit Error Value

To model failure in other cases, we adopt the concept of the explicit error, as presented by in counter-factual typing [24, 5].

We introduce the syntactic form \perp , which can be used in place of values, heads, etc. As an implementation detail, we annotate \perp with a string describing the context of failure, allowing us to recover useful information during error message generation.

When we encounter a flex-rigid or flex-flex equation with no solutions, such as in an occurs-check, we can assign \perp to the variable whose value we are trying to find, and proceed with unification as if a value had been found.

Moreover, when a value fails to typecheck, when generating its normal form, we replace the ill-typed portion with \perp . Replacing only the ill-typed portion of a term is made easy through our unified typing and normalization relation. By doing this, we prevent our algorithm from getting stuck where the unmodified algorithm would.

The modified rules for this can be seen in Figure 7. As presented, the rules are not syntax-directed, but in an implementation we avoid the (err-synth) rule unless no other rules apply. Since any term can be typed with normal form \perp , we add an extra condition to our definitional equality, so that different ill-typed terms are not treated as equal.

6.5 Correctness of Replay Graphs and Error-Tolerance

Here, we sketch the proof that the success or failure of unification is not changed by our replay graph and error-tolerant modifications.

We take as an assumption the correctness and uniqueness of solutions from the unmodified unification algorithm. That is, if $\Delta_1 \mapsto^* \Delta_n$ and $\Delta_1 \mapsto^* \Delta_k$, and Δ_n and Δ_k are both final, then either Δ_n and Δ_k are both solved containing equal solutions, or both represent failure (stuck or \perp). Correctness for similar algorithms is sketched by Abel and Pientka [15] and Gundry and McBride [22].

First, we state our main result:

Theorem 1. *Suppose Δ_n is a solved metacontext. Then $\Delta_1 \mapsto^* \Delta_n$ if and only if $\Delta_1 \hat{\mapsto}^* \Delta_n$ with replay graph G_n containing no error paths.*

To prove the “only if” direction, we first establish that our algorithm does not fail on satisfiable problems. This can be proved using straightforward induction. Intuitively, the result holds because our algorithm’s modifications only arise when unification fails.

Lemma 1. *If $\Delta_1 \mapsto^* \Delta_n$, where Δ_n is not \perp , then $\Delta_1 \hat{\mapsto}^* \Delta_n$.*

The final step in the “only if” direction is to prove that the resulting graph never contains an error when unmodified unification succeeds.

Lemma 2. Suppose $\Delta_1 \mapsto^* \Delta_n$ and $\Delta_1 \widehat{\mapsto}^* \Delta_n$ with replay graph \mathcal{G}_n , where Δ_n is a successful solution to Δ_1 . Let σ be the solution substitution implied by Δ_n . Then, if $\mathcal{V}(s)$ has a path to $\mathcal{V}(t)$, then $\sigma(s) =_{\beta\delta\eta} \sigma(t) : T$ for some type T .

Proof. By strong induction on the length of the path.

If the path has length one, then it is a single edge E . If the edge was added from a problem or a solution, then the result follows from the correctness of unmodified unification. Otherwise, the edge was added after a substitution in a raw term, meaning the equality holds by definition.

The inductive step is easily achieved: we divide any path of length $n > 1$ into a paths of length $n - 1$ and 1, apply our hypothesis to each sub-path, along with the transitivity of $=_{\beta\delta\eta}$.

□

Proving the “if” direction is slightly more involved, since our algorithm does behave differently when errors are encountered. We instead prove the contrapositive: any time the unmodified unification fails, the error tolerant algorithm also fails. We prove separate lemmas for the cases when unmodified produces \perp or gets stuck.

The unmodified algorithm produces \perp either by a rigid-rigid mismatch, or by an occurs-check/pruning failure. Our algorithm records these errors in the graph, or by assigning a variable \perp , giving us the following lemma.

Lemma 3. Suppose $\Delta_1 \mapsto^* \Delta_n$, where Δ_n is \perp . Then $\Delta_1 \widehat{\mapsto}^* \Delta'_m$ with replay graph \mathcal{G}_m , where one of the following holds:

- \mathcal{G}_m contains an error path.
- A solution in Δ_m contains \perp .

Finally, we must deal with the cases where unification fails not by producing \perp , but by reaching an unsolved state where no other rules apply. This happens when a typing or definitional-equality side-condition fails, or when no problems are in the pattern fragment. Since we produce a term containing \perp when typechecking fails, we obtain the following lemma, which is the final piece needed to prove our main result.

Lemma 4. Suppose $\Delta_1 \mapsto^* \Delta_n$, where Δ_n is stuck. Then $\Delta_1 \widehat{\mapsto}^* \Delta_n$ with replay graph \mathcal{G}_n , where one of the following holds:

- \mathcal{G}_n contains an error path
- A solution in Δ_n contains \perp
- Δ_m is stuck

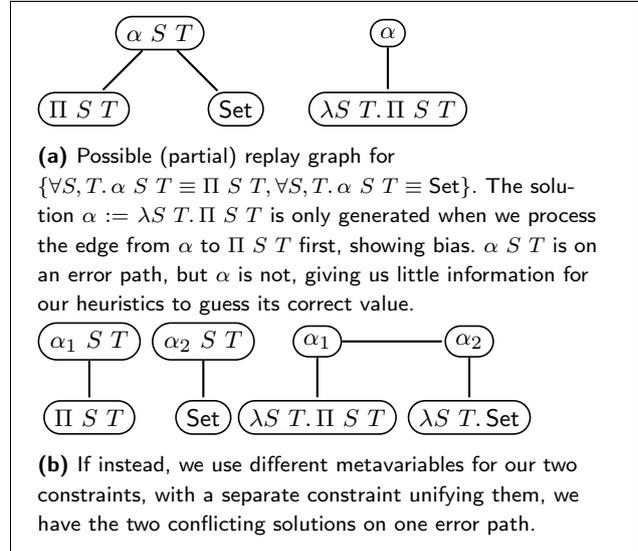


Fig. 8: Bias and a counter-factual solution

Additionally, these lemmas show that each specific error found using \mapsto is also found using $\widehat{\mapsto}$, since they behave identically until immediately before \mapsto reaches \perp .

7 Counter-Factual Unification

In the above sections, type graphs are used only to provide replays of unification, and do not direct the unification process. Thus, they still contain some bias: the order in which we solve constraints can affect the results.

We present *counter-factual unification*: a modification to unification which seeks to reduce bias by exploring solutions arising from subsets of the initial constraint-set. The main idea is that, when we find a solution for a variable, we proceed with unification both with the solution, and as if the solution had never been found. This allows us to see conflicting potential solutions, without being biased towards which one was generated first.

To accommodate this process with relative efficiency, we utilize the *choice calculus* [25] for compact representation of multiple constraint sets, as well as the modifications to unification needed for solving these. We then prove that our modified algorithm halts, and produces equivalent results to unmodified unification.

7.1 Motivation

While there are several potential sources of bias, a particularly problematic one is in substitutions. The order in

which constraints are solved in dynamic higher-order unification matters, and when a potential value is found, it is substituted into all other problems as the “actual” value, and future conflicting values are seen as rigid errors.

The inability of type graphs to handle this case arises from the need to unify arbitrary terms containing metavariables, particularly those containing function applications.

Consider the following unsatisfiable constraint set, whose replay graph is given in Figure 8:

$$\{P_1 = ?\forall S, T. \alpha \ S \ T \equiv \Pi \ S \ T, \quad P_2 = ?\forall S, T. \alpha \ S \ T \equiv \text{Set}\}$$

In normal unification, if we solve P_1 first, we define $\alpha := \lambda S \ T. \Pi \ S \ T$. We substitute the new value into P_2 , and after evaluation, get $\Pi \ S \ T \equiv \text{Set}$. This update to the value of $\alpha \ S \ T$ is recorded. However, in our type graph, our node for α will only have an edge to $\lambda S \ T. \Pi \ S \ T$. We have a path from $\Pi \ S \ T$ via $\alpha \ S \ T$ to Set , but the node for α itself occurs on no error paths. We are never able to see that $\lambda S \ T. \text{Set}$ is a potential solution for α .

The opposite problem happens if we begin with the second constraint, exposing *bias* in our solving procedure. Even with type-graphs, the order in which we solve constraints affects the error messages. Ideally, we want all conflicting values for α to be connected to it, so that we can present them as possible solutions, or use heuristics to determine which one is likely to be correct.

As a potential solution, suppose that we decomposed the above constraint set into the following:

$$\{P_1 = ?\forall S, T. \alpha_1 \ S \ T \equiv \Pi \ S \ T, \\ P_2 = ?\forall S, T. \alpha_2 \ S \ T \equiv \text{Set}, \quad P_3 = ?\alpha_1 \equiv \alpha_2\}$$

As we can see in Figure 8, we now obtain an error path containing both α_1 and α_2 .

We formalize the intuition behind this procedure, showing how to solve such problems. Specifically, we adapt the ideas from counter-factual typing [5], which in turn is based on the Choice Calculus [25]. These provide a method to counteract bias by examining solutions that arise if we had never given a variable its definition.

7.2 A Choice Calculus

7.2.1 Choice Operations and Terminology

In order to facilitate counter-factual solving, we need a way to compactly represent multiple possible solutions to a unification problem. To allow this, we augment our language of values with a form for *named choice expressions*: $t ::= \dots \mid C\langle t_1, t_2 \rangle$, where C is an identifier. If a term t

$$\begin{aligned} s, t, S, T & ::= \dots \mid C\langle s, t \rangle \mid \alpha_q \\ d, e & ::= \dots \mid C\langle d, e \rangle \\ E & ::= \dots \mid C\langle E, t \rangle \mid C\langle v, E \rangle \\ q & ::= \cdot \mid qC^L \mid qC^R \end{aligned}$$

$$\begin{aligned} C\langle s, t \rangle \cdot C\langle e_1, e_2 \rangle \bar{e}_i^i & \rightarrow C\langle s \cdot e_1, t \cdot e_2 \rangle \cdot \bar{e}_i^i \\ t \cdot C\langle e_1, e_2 \rangle \bar{e}_i^i & \rightarrow \text{hoist}(C, t) \cdot C\langle e_1, e_2 \rangle \bar{e}_i^i \\ C\langle s, t \rangle \cdot e \bar{e}_i^i & \rightarrow C\langle s, t \rangle \cdot \text{hoist}(C, e) \bar{e}_i^i \end{aligned}$$

$$\begin{aligned} \text{(meta-qualL)} & \frac{\Gamma \mid \Delta \vdash \alpha_q \rightsquigarrow \alpha_q \Rightarrow T}{\Gamma \mid \Delta \vdash \alpha_{qC} \rightsquigarrow \alpha_{qC^L} \Rightarrow C_L T} \\ \text{(meta-qualR)} & \frac{\Gamma \mid \Delta \vdash \alpha_q \rightsquigarrow \alpha_q \Rightarrow T}{\Gamma \mid \Delta \vdash \alpha_{qC^R} \rightsquigarrow \alpha_{qC^R} \Rightarrow C_R T} \\ \text{(ch-hoist1)} & \frac{\Gamma \mid \Delta \vdash C\langle s, t \rangle \rightsquigarrow u \Leftarrow \text{hoist}(C, T)}{\Gamma \mid \Delta \vdash C\langle s, t \rangle \rightsquigarrow u \Leftarrow T} \\ \text{(ch-hoist2)} & \frac{\Gamma \mid \Delta \vdash \text{hoist}(C, t) \rightsquigarrow u \Leftarrow C\langle S, T \rangle}{\Gamma \mid \Delta \vdash t \rightsquigarrow u \Leftarrow C\langle S, T \rangle} \\ \text{(ch-synth)} & \frac{\Gamma \mid \Delta \vdash s \rightsquigarrow u \Rightarrow S \\ \Gamma \mid \Delta \vdash t \rightsquigarrow v \Rightarrow T}{\Gamma \mid \Delta \vdash C\langle s, t \rangle \rightsquigarrow C\langle u, v \rangle \Rightarrow C\langle S, T \rangle} \\ \text{(ch-check)} & \frac{\Gamma \mid \Delta \vdash s \rightsquigarrow u \Leftarrow S \\ \Gamma \mid \Delta \vdash t \rightsquigarrow v \Leftarrow T}{\Gamma \mid \Delta \vdash C\langle s, t \rangle \rightsquigarrow C\langle u, v \rangle \Leftarrow C\langle S, T \rangle} \end{aligned}$$

Fig. 9: Choice syntax, semantics and typing

contains choices C_1, \dots, C_n , we say that these are the *dimensions* of the term, denoted by $\text{dim}(t)$.

In order to avoid conflicts between solutions in different choice dimensions, we qualify metavariables by the choices inside which they occur [24]. A qualifier string q is some (possibly empty) sequence of qualifiers, each of which is either C^L (left) or C^R (right), where C is the name of a choice. We use t_q as shorthand for adding the qualifier q to all metavariables in t .

For each choice identifier C , we define left and right projections, C_L and C_R . These map every occurrence of choice C to their left or right sides, respectively, while stripping metavariables of any C qualifiers. We allow these projections to be applied to arbitrary terms, constraints, and metacontexts, where all occurrences of a choice are replaced with the left or right variant respectively. Similarly, we define a projection **Factual** which maps all choices to their left-hand side.

One final operation that we will make use of is *hoisting*. This allows us to take a term containing multiple choices in the same dimension, and lift them to the top level. When we hoist a term t with respect to choice C , denoted $\text{hoist}(C, t)$ we produce the term $C\langle C_L t, C_R t \rangle$. Note that hoist , C_L , C_R and **Factual** are operations in the metalanguage, rather than constructs in the language itself.

For example, consider the following term:

$$t = C_1\langle C_2\langle x, y \rangle, z \rangle$$

The projection $C_{1L} t$ gives us $C_2\langle x, y \rangle$, and $C_{1R} t$ gives z . Similarly, $C_{2L} t$ gives $C_1\langle x, z \rangle$, and $C_{2R} t$ gives $C_1\langle y, z \rangle$. To get the left side of all choices, we can write **Factual** t to obtain x . To bring C_2 as the top level choice, we use $\text{hoist}(C_2, t)$ to obtain the equivalent expression $C_2\langle C_1\langle x, z \rangle, C_1\langle y, z \rangle \rangle$. Hoisting C_1 is a valid operation, but returns t unchanged since C_1 is already the top-level choice.

7.2.2 Semantics and Typing

The semantics and typing rules for choice are in Figure 9. To apply an eliminator to a head, where both have a top-level choice in dimension C , we simply return the choice between applying the left eliminator to the left head and the right eliminator to the right head. If only one of the head and eliminator is a choice, or they are choices in different dimensions, we can hoist to move the same choice to the top level.

The type rules work in much the same way. In (meta-qualL) and (meta-qualR), we synthesize the type for a qualified type variable by finding the type for the unqualified

variable, and projecting left or right according to the qualifier tag. The remaining rules account for choices in types and terms. There are two steps to checking. First, we must ensure the term and the type we are checking it against have the same top-level choice. We do this with (ch-hoist1) and (ch-hoist2), using **hoist** to bring any nested choices to the top-level. Then, to check a choice expression against a choice type in (ch-check), we simply check each dimension of the choice separately. To synthesize the type for a choice expression, we synthesize a type for each dimension, then pair these with the expression's choice label.

Intuitively, choices represent the set of terms obtained from all possible combinations of projections on each dimension. There are 2^n constraints represented by a term containing n distinct choice labels, assuming each choice has two distinct variants. The counting is more complicated when choice labels are nested, but a potentially exponential set of values is in this way compactly represented.

7.3 Counter-Factual Solving

In Figure 10 we provide the definition for our counter-factual unification rewrite relation, denoted by $\overset{\sim}{\rightarrow}$. We assume that all rules not explicitly listed are identical to $\overset{\sim}{\rightarrow}$.

The main idea of counter-factual solving is that whenever unmodified unification generates a substitution $\alpha := t$, we instead generate $\alpha := C\langle t, \alpha' \rangle$, where C and α' are fresh, so that the solver can also proceed as if no value were ever given to α . We refer to t as the *factual case*, and to α' as the *counter-factual case*, since they present the solutions for whether the constraint was or was not present.

Traditional counter-factual typing is defined as part of the typing judgement. However, due to the complexities introduced by dependent types, we instead use choice only during the unification process.

Defining Metavariables

When we find a solution $\alpha := t$ in a problem P , we instead generate $C\langle t_{CL}, \alpha' \rangle$ as a solution, where α' is a fresh metavariable. Here, t_{CL} is t with the qualifier C^L added to all metavariables occurring within t . This ensures that, if solutions to α' contain metavariables in t , their solutions will not conflict.

Solving Equations with Choice

An equation with a choice point represents a set of separate equations, which we would like to solve separately, without solutions to one equation affecting the others. This problem is solved by considering qualified type variables as distinct

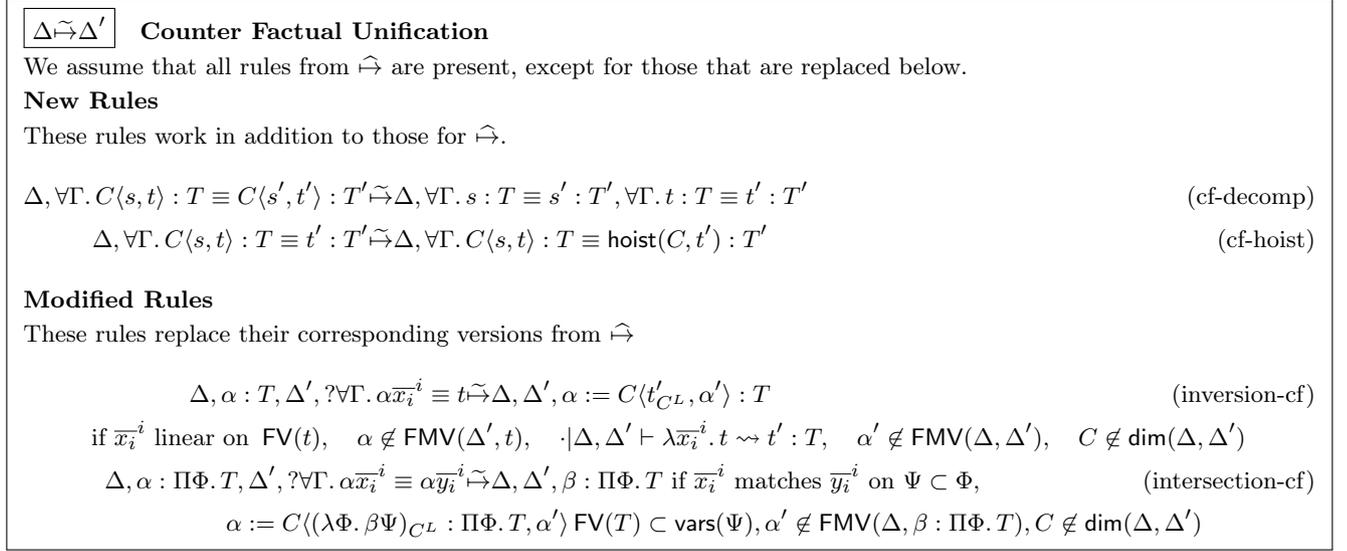


Fig. 10: Counter-factual unification rules

unless they are identical variables with identical qualifiers. Solutions for qualified variables are generated, and the full solution for variable α can be reconstructed from the qualified solutions through *completion* [26], where it is assigned a value whose choices correspond to the choices of each qualified solution. However, our treatment of choices within replay graphs means that we do not need to perform completion to generate error messages. We elaborate on this in Section 7.4.

Solving equations involving choice can be achieved through manipulating those choices. The equation $C\langle s, t \rangle \equiv C\langle s', t' \rangle$ is easily solved by decomposition into $s \equiv s', t \equiv t'$. When only one side is a choice in dimension C at the top level, we can apply **hoist**, which will move any occurrences of C to the top level. If one side is a term t containing no choices, **hoist**(C, t) produces $C\langle t, t \rangle$. Note that we must hoist here, but we wish to hoist only when necessary, since it can duplicate large parts of the term being hoisted.

7.3.1 Performance

When we apply counter-factual solving, we see an exponential explosion in the number of problems and the size of the context, in what is already a potentially slow algorithm. While the choice calculus helps keep this number smaller, most choices will ultimately be pointless, as we will find equal values for both sides of choice expressions.

In order to facilitate a practical solving algorithm, when we split a constraint involving a choice, as described above, we mark newly generated equations for the right-

side of the choice as pending on the failure of the left-hand side. Thus, if the unification problems involving the choice all succeed, we never explore the counter-factual cases.

7.3.2 Redundancy

Another practical issue with counter-factual unification is that our type graphs are now redundant. Since the factual case behaves as unmodified unification does, each error is reflected twice in our type graph: once for the error in the factual case, and once in the disagreement between the factual and counter-factual cases.

To avoid this redundancy, an implementation can keep in its context the string of choice dimensions that have been currently assumed. For example, when solving $C_1\langle C_2\langle T_1, T_2 \rangle, T_3 \rangle \equiv S$, we would store the string $C_1\overline{C_2}$ when solving the equation $T_2 \equiv S$. Then, in our graph, we make each generated edge dependent on the solution that first introduced the choices stored in the context. With our example above, if C_1 was a fresh choice generated by the solution $\alpha := C_1\langle S', \alpha' \rangle$, and C_2 from $\beta := C_2\langle S'', \beta' \rangle$, then the edge $\{T_2, S\}$ would be dependent on the edges $\{\alpha, S'\}$ and $\{\beta, S''\}$.

The intuition behind this is that, when diagnosing an error, if we delete edge from the factual case of a solution, we should also delete the edges that were made under the assumption that the factual case held. Each intermediate constraint will still be represented in the graph, but only in the counter-factual case for the deleted edge.

7.4 Counter-Factual Replay Graphs

The use of counter-factual typing helps generate potential solutions for ill-typed metavariables. However, we still wish to take advantage of the information available to us from type graphs, and the heuristics for diagnosing error locations.

Our solution is simple: when we generate a definition $\alpha := C\langle t, \alpha' \rangle$, we add an equality edge between α and α' . For other occurrences of choice, we simply add an undirected edge between the vertices for the two sides of the choice. The intuition behind this is that, when a program is well typed, both dimensions of a choice for a metavariable's solution should agree. Disagreements between the factual and counter-factual cases become error paths in our program.

Additionally, we can make use counter-factual solving in a repair heuristic. Whenever we define $\alpha := C\langle t, \alpha' \rangle$, we have an edge from α to t , and one from α to α' . If we remove the edge from α to t (as well as its dependent edges), and the new graph has fewer inconsistencies, then we can suggest the value of α' as a likely fix for the value of α .

7.5 Correctness

Here we establish the relationship between counter-factual unification and our error-tolerant algorithm. Our main result is as follows:

Theorem 2. $\Delta_1 \widehat{\mapsto^*} \Delta_m$ and $\Delta_1 \widetilde{\mapsto^*} \Delta'_n$, where Δ_m and Δ'_n are both final, then Δ_m is solved if and only if Δ'_n is.

We first note that we can simulate error-tolerant unification in our counter-factual system, This can be proved by induction, utilizing the fact that the only modified rules keep the original result in the left-hand side of the choice. The counter-factual run may take more steps, since some steps are required to decompose choices.

Lemma 5. If $\Delta_1 \widehat{\mapsto^*} \Delta_m$, then $\Delta_1 \widetilde{\mapsto^*} \Delta'_n$, where $m \leq n$ and $\text{Factual}(\Delta'_n) = \Delta_m$.

Conversely, we can recover an error-tolerant run from any counter-factual run. The proof is similar to Lemma 5, with the added detail that the (unif-choice) rule produces changes that are completely erased by **Factual**.

Lemma 6. If $\Delta_1 \widetilde{\mapsto^*} \Delta'_n$, then $\Delta_1 \widehat{\mapsto^*} \Delta_m$, where $m \leq n$ and $\text{Factual}(\Delta'_n) = \Delta_m$.

Finally we strengthen Theorem 2 to show that counter-factual solving terminates whenever normal unification does.

Theorem 3. If $\Delta_1 \widehat{\mapsto^*} \Delta_n$, where Δ_n is solved or stuck, then there exists some m such that $\Delta_1 \widetilde{\mapsto^*} \Delta'_m$, where Δ'_m is either solved or stuck.

Proof. Gundry and McBride [22] give an intuition that \mapsto always progresses towards a solution, and we have shown that $\widehat{\mapsto}$ will reach a solution or failure in finitely many steps whenever \mapsto does. Thus, we assume a well-founded ordering \prec such that if $\Delta \widehat{\mapsto} \Delta'$, then either Δ and Δ' differ only in permutations and symmetry, or $\Delta' \prec \Delta$.

We define a new partial order \sqsubseteq as the smallest partial order where $\Delta' \sqsubseteq \Delta$ when any of the following hold:

$$\Delta' \prec \Delta \quad (1)$$

$$C_L \Delta' \prec \Delta \text{ and } C_R \Delta' \prec \Delta \text{ for some } C \quad (2)$$

$$\Delta' \text{ is } \Delta \text{ with a choice hoisted outwards} \quad (3)$$

\sqsubseteq is well founded. In any descending chain, the number of choice dimensions may increase only if each underlying dimension is strictly descending with respect to \prec . So the number of dimensions we can add in a descending \sqsubseteq chain is limited by the length of descending \prec chains, which are always finite. Likewise, choices can only be hoisted outwards finitely many times.

Finally, we show that if $\Delta \widetilde{\mapsto} \Delta'$, then $\Delta' \sqsubseteq \Delta$, or they are equal up to permutation and symmetry. This is immediate for unmodified rules by (1). For (inversion-cf) and (intersection-cf), we introduce a dimension C for our solution, but replace a problem with solution in each dimension, so our order is respected by (2). For (ch-decomp) we are smaller by (1) because we decompose one problem into two structurally smaller ones ($C\langle s, t \rangle$ is structurally larger than both s and t). For (ch-hoist) we are smaller by (3) because a choice is hoisted outwards. \square

8 Results and Discussion

In order to evaluate our techniques, we combined the implementations of Helium, LambdaPi, and Gundry-McBride Unification. A few Helium heuristics, such as the application and permutation heuristics, were transferred to our system, along with our index-permutation heuristic specific to dependent types. The source code of the implementation is publicly available [27].

We present some simple programs containing type errors, and compare our generated error messages with those from roughly equivalent programs in Agda and Idris. When relevant, we include our generated messages both with and without counter-factual solving, to highlight where it causes improvement. For brevity, we omit the code for the Agda and Idris versions.

An issue with solving is that several errors are often identified, where only one should be. For the sake of these examples, we simplify the messages presented to a single error, in order to highlight the repair heuristics of our system. We discuss the problems with the full list of errors in Section 8.2.

8.1 Results

8.1.1 Too Many Arguments

If we give a function too many arguments, we can give a helpful hint informing the user of the expected number. From the replay graph, we can look at the context of the error and find the entire function type, seeing that it takes three arguments. Both Agda and Idris only report the error for the partially-applied function, and Agda does not even warn the user that `Nat` is not a function type.

Unfortunately, due to limitations in our version of the application heuristic, metavariables are left in the reported expected type.

```
let myFun = (\ a x y -> x) :: forall (a :: *) . a -> a
↳ -> a
let myApp = myFun _ 0 1 2
-- Mismatch in type of myFun _ (0 :: Nat) (1 :: Nat)
↳ (2 :: Nat)
-- ?A1 -> ?A2 /= Nat
-- HINT: Function expected at most 3 arguments,
↳ but you gave 4
```

```
Agda: Nat should be a function type, but it isn't
when checking that Zero Zero are valid arguments to a
↳ function of type Nat
Idris: When checking right hand side of myApp with
↳ expected type Nat
When checking an application of function myFun:
Type mismatch between Nat (Type of Zero) and _ -> _
↳ (Is Zero applied to too many arguments?)
```

8.1.2 Too Few Arguments

When too few arguments are given, our repair heuristics are able to suggest both the types and positions of arguments which must be added in order to create a well-typed function call. This is done by repeatedly adding arguments with metavariable types to the application in the replay

graph, stopping either when the type graph becomes consistent, or when the return type is no longer an arrow type. Agda and Idris only inform the user that the actual type of the expression is a function type, describing its mismatch with the result type.

```
let myFun = (\ x y -> x) :: Nat -> Nat -> Nat
let myApp = (myFun 0) :: Nat
-- Mismatch in type of result of application myFun
↳ (0 :: Nat)
-- Nat /= Nat -> Nat
-- HINT: Function expected 2 arguments, but you
↳ gave 1.
-- Try ("myFun") (Zero) (x2) where x2 :: Nat
```

```
Agda: Nat -> Nat !=< Nat of type Set
when checking that the expression myFun _ Zero has
↳ type Nat
Idris: When checking right hand side of myApp with
↳ expected type Nat
Type mismatch between a -> a (Type of myFun a _) and
↳ Nat (Expected type)
```

8.1.3 Arguments in the Wrong Order

As with too few arguments, our repair heuristics can use a similar procedure to try different permutations of arguments, suggesting the one that is likely to resolve the error.

```
let myFun = (\ _ x y -> y) :: forall (a :: *) . a ->
↳ Nat -> Nat
let myApp = (myFun _ 0 (Nil Nat))
-- Mismatch in type of myFun _ (0 :: Nat) (Nil Nat)
-- Nat /= Vec Nat 0
-- HINT: Function arguments in the wrong order.
-- Try ("myFun") _ (Nil Nat) (0)
```

When counter-factual solving is enabled, however, our heuristics fail in this case.

```
-- Mismatch in type of result of application Nil Nat
-- Vec Nat Zero /= Nat
```

Agda and Idris report a mismatch, but provide no hint that the arguments should be swapped.

```
Agda: Vec Nat Zero !=< Nat when checking that the
↳ expression Nil Nat has type Nat
Idris: When checking right hand side of myApp with
↳ expected type Nat
When checking an application of function myFun:
Type mismatch between Vec a Zero (Type of []) and Nat
↳ (Expected type)
```

8.1.4 Dependent Type Error

Here we show the error message arising from a classic, dependent-type specific mistake: reversing the indices of

an Eq value. This example can be fixed by reversing the type signature, or applying a proof of equality’s symmetry.

We are able to apply an *isomorphism heuristic*, a variant of the permutation heuristic that permutes dependent type indices. This allows us to see that the two conflicting types are isomorphic to one another, prompting the user to rearrange them. However, we can only do this when counter-factual solving is enabled.

The error message without counter-factual solving is unsatisfactory, generating a false positive with the application heuristic. As we described when motivating counter-factual unification, the type graph lacks an error path between `(plus (Succ x) 0)` and `(Succ x)`, containing only a path between the Eq types containing these as indices.

For brevity, we use `plus` in place of its body in our error messages.

```
let plus =
natElim ( \ _ => Nat -> Nat ) ( \ n => n ) ( \ p rec
  ↪ n -> Succ (rec n) )
assume pNPlus0isN :: forall n :: Nat . Eq Nat (plus n
  ↪ 0) n
let succPlus = (\n -> pNPlus0isN (Succ n)) :: forall
  ↪ n :: Nat . Eq Nat (Succ n) (plus (Succ n) 0)
--Mismatch in type of result of application pNPlus0isN
  ↪ (Succ x)
--Eq Nat (Succ x) (Succ (plus (Succ n) 0)) /= Eq Nat
  ↪ (Succ (plus (Succ n) 0)) (Succ x)
--
  HINT: Function arguments in the wrong order.
  ↪ Try ("pNPlus0isN") (Succ x)
```

If we enable counter-factual solving, we get a path between `(plus (Succ x) 0)` and `(Succ x)`, since one is the expected type in the factual case, and the other is expected in the counter-factual case. This allows our heuristics to switch which edges connect to these, allowing for the permutation hint to be suggested.

```
-- HINT: Rearrange arguments to match
-- Eq Nat (Succ x) (plus (Succ x) 0) to Eq Nat
  ↪ (plus (Succ x) 0) (Succ x)
```

Agda and Idris report the mismatch but are unaware that a swap would fix it.

```
Agda: natElim (\_=> Nat -> Nat) (\n1-> n1) (\p rec
  ↪ n2-> Succ (rec n1)) n Zero != n of type Nat
when checking that the expression pNPlus0isN (Succ n)
  ↪ has type
  Eq Nat (Succ n) (plus (Succ n) Zero)
Idris: When checking right hand side of succPlus with
  ↪ expected type
(n : Nat) -> Eq Nat (Succ n) (plus (Succ n) Zero)
When checking argument n to pNPlus0isN:
Type mismatch between Succ (natElim (\_ => Nat -> Nat)
  ↪ (\n1 => n1)
  (\p => \rec => \n5 => Succ (rec n5)) n Zero)
  ↪ (Inferred value) and Succ n (Given value)
```

8.2 Limitations

While we are able to provide helpful hints in many cases, the system is not without its flaws. Our results are promising, but the issues below will require solutions before our techniques are ready for integration into a mainstream language.

Error Redundancies

Our compiler sometimes reports the same message multiple times. Specifically, the actual messages generated have redundancies with counter-factual solving in Sections 8.1.2 and 8.1.3, and with or without without counter-factual solving in Section 8.1.4.

This is unsurprising with counter-factual solving: since we fully simulate normal unification in the factual case, each error will be recorded twice: once in the factual case, and once in the disagreement between the factual and counter-factual case. The strategy of Section 7.3.2 should be helpful in alleviating these redundancies.

The causes are more subtle for the case without counter-factual solving. In Section 8.1.4, our compiler actually reports a conflict between each combination of the following four types:

$$\text{Eq } (plus (Succ x) 0) (Succ x)$$

$$\text{Eq } (Succ x) (plus (Succ x) 0)$$

$$\text{Eq } (Succ x) (Succ x)$$

$$\text{Eq } (plus (Succ x) 0) (plus (Succ x) 0).$$

This is because internally, it has $\text{Eq } \alpha \beta$, where α and β are metavariables each with edges to `(Succ x)` and `(plus (Succ x) 0)`. Further development of heuristics should help in reducing this kind of redundancy.

Dependent Edges in Repair Heuristics

Our compiler uses replay graphs to generate derived edges from initial constraints. However, many heuristics involve removing and adding edges in the graph. It is easy to add implicit edges from constructor applications. However, the situation is complicated for dependent edges generated during unification. When modifying the graph in a heuristic, removing the edges for an initial edge deletion is simple, but generating dependent edges when new edges are added is not. Re-running unification may be too costly for practical purposes. Because of this, some heuristics generate “false positives,” where a suggestion is given that will not actually resolve the type errors, because dependent edges cause inconsistencies that are not immediately apparent in

the graph. This is why the example without counter-factual typing gave an inaccurate hint in Section 8.1.4.

Controlling Evaluation

The error messages our compiler generates are often long and unwieldy. This is because Gundry and McBride’s implementation, and hence ours, is aggressively strict in its name lookup strategy. As a result, names are prematurely substituted for their values. The examples above involving *plus* would be much more readable if the actual printed used *plus* in place of its body, as we present it.

The issue of lazy versus strict evaluation, and of choosing when to replace identifiers by their values, seems critical for the generation of readable error messages, and we suspect addressing this issue will be a project unto itself.

8.3 Future Work

In addition to resolving the above limitations, we highlight a few more potential research topics.

Heuristics

We have presented a framework, which allows for the analysis of type-errors using heuristics on type graphs. While we have adapted a few Helium heuristics to our framework, there is room for heuristics specifically targeted at dependent types. A Helium-style collection of heuristics could be manually collected, or future work could incorporate the Bayesian heuristics from SHErrLoc [28].

Performance

Initially, performance is not a major concern: since dependent types require annotations, each declaration can be type-checked separately, so our techniques could be limited to the specific module, or even specific binding group, that the programmer is currently writing.

That said, the unification algorithm which we adapted was, according to the authors, not tuned for performance, with much needless iteration through the context. Similarly, the use of counter-factual solving adds exponential slowdown to an already slow algorithm. While we can keep the number of counter-factual cases low by only using choice after an error is encountered, a more sophisticated algorithm could yield performance improvements.

Switch combinators were developed for Helium [4] to address the computational costs associated with type-graph analysis. Possible future work could examine how

these would need to be changed to be adapted to our system.

Alternate Constraint Solvers

While our algorithm is based on Gundry and McBride’s presentation of unification, there are several different higher order unification algorithms [15, 19]. Gundry-McBride unification was ideal for exploratory research, but other algorithms could be faster, or able to solve a wider set of problems. The concepts we introduce, such as replay-graphs and counter-factual solving, can likely be applied to more sophisticated constraint solvers.

9 Related Work

9.1 Unification Algorithms

Huet [29] showed that higher order unification was undecidable in general, but later provided a semi-decision procedure for it [30]. The original pattern unification algorithm was presented by Miller [14]. This was later extended by Abel and Pientka [15] to accommodate dependent pairs and records, as well as to be *dynamic*, allowing problems outside the pattern fragment so long as they are eventually removed by substitutions from solving variables. Gundry and McBride [22] presented a version of unification tailored towards a practical implementation. The algorithm we present is a direct modification of theirs, along with the version in the thesis of Gundry [21].

A unification algorithm for Coq has been developed to account for universe polymorphism and overloading, as well as to incorporate heuristics to solve some cases outside of the pattern fragment [19]. In Agda, the pattern matching and unification were modified to accommodate univalent theories [31], and a new unification algorithm has been added that models unifiers as dependently typed equivalences [32, 23].

9.2 Error Message Improvement

To our knowledge, our work is the first to specifically explore error message reporting for dependent types. However, the topic has been studied extensively for Hindley-Milner and Object-Oriented languages.

Reporting Strategies

Haack and Wells [8] introduce the concept of an *error slice*: a set of program points which contribute to the error. This was one of the first constraint-based methods for error reporting, and is based around the idea that multiple locations should be reported, since the true location of an error can depend on the programmer’s desired semantics.

The idea of presenting multiple error locations is extended by Chen and Erwig [5] through a technique known as *counter-factual* typing. In counter-factual typing, the checker examines atomic expressions and determines what type the expression would need to have in order for the program to typecheck. This, in turn, was an extension of the work on variational lambda calculus and error-tolerance [26, 24]. The counter-factual unification and error-tolerant typing we present are an adaptation of this work to dependent types.

Helium

Helium [2, 3, 4] is an alternate Haskell compiler, which aimed to improve the quality of error messages, particularly for beginners. At its core was a type-inference algorithm based on the concept of *constraint-generation*, where constraints are emitted, rather than having a single substitution gradually grow as in Algorithm W [33]. The various strategies for solving these constraint sets can emulate classical inference algorithms [34], as well as allowing for heuristic analysis. A more advanced technique Helium used was the construction of *constraint graphs*, creating a global representation of the entire type inference problem to be heuristically analyzed. Our replay graphs are a direct adaptation of Helium’s type graphs, and our implementation draws heavily from the public Helium code base [35].

Bias

Much of the work on error diagnosis seeks to avoid bias. In Helium, type graph help avoid bias within a binding group, but not between binding groups. Counter-factual typing avoids the latter by making it possible to blame an early binding group if a later binding group provides the information to pinpoint the actual mistake, essentially by not making us commit to a particular choice early on in the program. The work by Pavlinovic, King, and Wies [36] takes a different approach which seems to scale much better: when a binding group contains a type error, definitions within the same module are inlined until the error location is *not* simply the location of a particular identifier defined earlier. This may mean that for that binding group the inference process is re-run multiple times, but in practice, a few inlining steps suffice.

SHErrLoc

An alternate approach to the constraint-based system of Helium is implemented in *SHErrLoc* [28]. Its constraints are over arbitrary partial orders, and is generic enough to describe Hindley Milner typing, as well as information-flow and data-flow analyses. Constraint solving uses *context-free reachability*. This approach can typecheck more advanced systems, such as typeclasses and GADTs [9]. However, it treats the axioms from type-level functions as pre-existing, and it is thus ill-suited to languages with unified types and terms and arbitrary type-level computations.

10 Conclusion

In this work, we have identified the challenges with adapting known functional error-message generation to dependently-typed languages. With replay graphs and counter-factual solving, we have taken the first steps towards improving error messages for dependent-types. The techniques were implemented for a small language, and initial evaluation showed that helpful hints could be generated for simple examples.

While our approach has limitations, we have shown that it is possible to use more advanced error generation techniques with dependent types. By identifying possible approaches, as well as challenges, for dependent type error generation, we hope to provide a solid foundation for future research on the ease of use of dependently-typed languages.

References

- [1] Titus Barik et al. “Do Developers Read Compiler Error Messages?” In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE ’17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 575–585. ISBN: 978-1-5386-3868-2. DOI: 10.1109/ICSE.2017.59. URL: <https://doi.org/10.1109/ICSE.2017.59>.
- [2] Bastiaan Heeren, Jurriaan Hage, and S Doaitse Swierstra. “Constraint based type inferencing in Helium.” In: *Intermediate Applications of Constraint Programming (ACP)* (2003), p. 57.
- [3] Jurriaan Hage and Bastiaan Heeren. “Implementation and Application of Functional Languages: 18th International Symposium, IFL 2006, Budapest, Hungary, September 4-6, 2006, Revised Selected Papers.” In: ed. by Zoltán Horváth, Viktória Zsóka, and Andrew Butterfield. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. Chap. Heuristics for Type Error Discovery and Recovery, pp. 199–216. ISBN: 978-3-540-74130-5. DOI: 10.1007/978-3-540-74130-5_12. URL: http://dx.doi.org/10.1007/978-3-540-74130-5_12.

- [4] Bastiaan J Heeren. “Top quality type error messages.” In: *IPA Dissertation Series*; (2005).
- [5] Sheng Chen and Martin Erwig. “Counter-factual Typing for Debugging Type Errors.” In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: ACM, 2014, pp. 583–594. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535863. URL: <http://doi.acm.org/10.1145/2535838.2535863>.
- [6] Ulf Norell. “Dependently Typed Programming in Agda.” In: *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*. TLDI '09. Savannah, GA, USA: ACM, 2009, pp. 1–2. ISBN: 978-1-60558-420-1. DOI: 10.1145/1481861.1481862. URL: <http://doi.acm.org/10.1145/1481861.1481862>.
- [7] Edwin Brady. “Idris, a general-purpose dependently typed programming language: Design and implementation.” In: *Journal of Functional Programming* 23.5 (2013), 552–593. DOI: 10.1017/S095679681300018X.
- [8] Christian Haack and J.B. Wells. “Type error slicing in implicitly typed higher-order languages.” In: *Science of Computer Programming* 50.1–3 (2004). 12th European Symposium on Programming (ESOP 2003), pp. 189–224. ISSN: 0167-6423. DOI: <http://dx.doi.org/10.1016/j.scico.2004.01.004>. URL: <http://www.sciencedirect.com/science/article/pii/S016764230400005X>.
- [9] Danfeng Zhang et al. “SHerrLoc: A Static Holistic Error Locator.” In: *ACM Trans. Program. Lang. Syst.* 39.4 (Aug. 2017), 18:1–18:47. ISSN: 0164-0925. DOI: 10.1145/3121137. URL: <http://doi.acm.org/10.1145/3121137>.
- [10] Jun Yang et al. “Improved Type Error Reporting.” In: *Proceedings of the 12th International Workshop on Implementation of Functional Languages*. Vol. 2011. IFL '00. RWTH Aachen: Springer Verlag, September 2000, pp. 71–86.
- [11] /u/GNUlinuxProgrammer. */r/agda: R x → x == y → R y*. https://www.reddit.com/r/agda/comments/7lq44q/r_x_x_y_r_y/. Accessed: 2018-06-14.
- [12] Kevin Watkins et al. “A Concurrent Logical Framework I: Judgments and Properties.” In: CMU-CS-02-101. School of Computer Science, Carnegie Mellon University, 2003.
- [13] Per Martin-Löf. “About Models for Intuitionistic Type Theories and the Notion of Definitional Equality.” In: *Proceedings of the Third Scandinavian Logic Symposium*. Ed. by Stig Kanger. Vol. 82. Studies in Logic and the Foundations of Mathematics. Elsevier, 1975, pp. 81–109. DOI: [https://doi.org/10.1016/S0049-237X\(08\)70727-4](https://doi.org/10.1016/S0049-237X(08)70727-4). URL: <http://www.sciencedirect.com/science/article/pii/S0049237X08707274>.
- [14] Dale Miller. “Unification Under a Mixed Prefix.” In: *J. Symb. Comput.* 14.4 (Oct. 1992), pp. 321–358. ISSN: 0747-7171. DOI: 10.1016/0747-7171(92)90011-R. URL: [http://dx.doi.org/10.1016/0747-7171\(92\)90011-R](http://dx.doi.org/10.1016/0747-7171(92)90011-R).
- [15] Andreas Abel and Brigitte Pientka. “Higher-Order Dynamic Pattern Unification for Dependent Types and Records.” English. In: *Typed Lambda Calculi and Applications*. Ed. by Luke Ong. Vol. 6690. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 10–26. ISBN: 978-3-642-21690-9. DOI: 10.1007/978-3-642-21691-6_5. URL: http://dx.doi.org/10.1007/978-3-642-21691-6_5.
- [16] Ulf Norell. “Towards a practical programming language based on dependent type theory.” PhD thesis. Chalmers University of Technology, 2007. URL: <http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf>.
- [17] Iliano Cervesato and Frank Pfenning. “A Linear Spine Calculus.” In: *Journal of Logic and Computation* 13.5 (2003), pp. 639–688. DOI: 10.1093/logcom/13.5.639. eprint: <http://logcom.oxfordjournals.org/content/13/5/639.full.pdf+html>. URL: <http://logcom.oxfordjournals.org/content/13/5/639.abstract>.
- [18] Robert Harper, Furio Honsell, and Gordon Plotkin. “A Framework for Defining Logics.” In: *J. ACM* 40.1 (Jan. 1993), pp. 143–184. ISSN: 0004-5411. DOI: 10.1145/138027.138060. URL: <http://doi.acm.org/10.1145/138027.138060>.
- [19] Beta Ziliani and Matthieu Sozeau. “A comprehensive guide to a new unifier for CIC including universe polymorphism and overloading.” In: *Journal of Functional Programming* 27 (2017), e10. DOI: 10.1017/S0956796817000028.
- [20] Andres Löf, Conor McBride, and Wouter Swierstra. “A Tutorial Implementation of a Dependently Typed Lambda Calculus.” In: *Fundam. Inf.* 102.2 (Apr. 2010), pp. 177–207. ISSN: 0169-2968. URL: <http://dl.acm.org/citation.cfm?id=1883634.1883637>.
- [21] Adam Gundry. “Type Inference, Haskell and Dependent Types.” PhD thesis. University of Strathclyde, 2013. URL: <http://adam.gundry.co.uk/pub/thesis/thesis-2013-12-03.pdf>.
- [22] Adam Gundry and Conor McBride. “A tutorial implementation of dynamic pattern unification.” In: *Unpublished draft* (2013). URL: <http://adam.gundry.co.uk/pub/pattern-unify/pattern-unification-2012-07-10.pdf>.
- [23] Jesper Cockx and Dominique Devriese. “Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory.” In: *Journal of Functional Programming* 28 (2018), e12. DOI: 10.1017/S095679681800014X.
- [24] Sheng Chen, Martin Erwig, and Eric Walkingshaw. “An Error-tolerant Type System for Variational Lambda Calculus.” In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ICFP '12. Copenhagen, Denmark: ACM, 2012, pp. 29–40. ISBN: 978-1-4503-1054-3. DOI: 10.1145/2364527.2364535. URL: <http://doi.acm.org/10.1145/2364527.2364535>.
- [25] Martin Erwig and Eric Walkingshaw. “The Choice Calculus: A Representation for Software Variation.” In: *ACM Trans. Softw. Eng. Methodol.* 21.1 (Dec. 2011), 6:1–6:27. ISSN: 1049-331X. DOI: 10.1145/2063239.2063245. URL: <http://doi.acm.org/10.1145/2063239.2063245>.
- [26] Sheng Chen, Martin Erwig, and Eric Walkingshaw. “Extending Type Inference to Variational Programs.” In: *ACM Trans. Program. Lang. Syst.* 36.1 (Mar. 2014), 1:1–1:54. ISSN: 0164-0925. DOI: 10.1145/2518190. URL: <http://doi.acm.org/10.1145/2518190>.
- [27] Joseph Eremondi. *GitHub Repository: lambda-pi-constraint, tag thesis-final*. <https://github.com/JoeyEremondi/lambda-pi-constraint>. 2016.
- [28] Danfeng Zhang et al. “Diagnosing Type Errors with Class.” In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI

2015. Portland, OR, USA: ACM, 2015, pp. 12–21. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2738009. URL: <http://doi.acm.org/10.1145/2737924.2738009>.
- [29] Gerard P. Huet. “The undecidability of unification in third order logic.” In: *Information and Control* 22.3 (1973), pp. 257–267. ISSN: 0019-9958. DOI: [https://doi.org/10.1016/S0019-9958\(73\)90301-X](https://doi.org/10.1016/S0019-9958(73)90301-X). URL: <http://www.sciencedirect.com/science/article/pii/S001999587390301X>.
- [30] G.P. Huet. “A unification algorithm for typed λ -calculus.” In: *Theoretical Computer Science* 1.1 (1975), pp. 27–57. ISSN: 0304-3975. DOI: [https://doi.org/10.1016/0304-3975\(75\)90011-0](https://doi.org/10.1016/0304-3975(75)90011-0). URL: <http://www.sciencedirect.com/science/article/pii/0304397575900110>.
- [31] Jesper Cockx, Dominique Devriese, and Frank Piessens. “Pattern Matching Without K.” In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: ACM, 2014, pp. 257–268. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628139. URL: <http://doi.acm.org/10.1145/2628136.2628139>.
- [32] Jesper Cockx, Dominique Devriese, and Frank Piessens. “Unifiers As Equivalences: Proof-relevant Unification of Dependently Typed Data.” In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. Nara, Japan: ACM, 2016, pp. 270–283. ISBN: 978-1-4503-4219-3. DOI: 10.1145/2951913.2951917. URL: <http://doi.acm.org/10.1145/2951913.2951917>.
- [33] Robin Milner. “A theory of type polymorphism in programming.” In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. ISSN: 0022-0000. DOI: [http://dx.doi.org/10.1016/0022-0000\(78\)90014-4](http://dx.doi.org/10.1016/0022-0000(78)90014-4). URL: <http://www.sciencedirect.com/science/article/pii/0022000078900144>.
- [34] Jurriaan Hage and Bastiaan Heeren. “Strategies for Solving Constraints in Type and Effect Systems.” In: *Electron. Notes Theor. Comput. Sci.* 236 (Apr. 2009), pp. 163–183. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2009.03.021. URL: <http://dx.doi.org/10.1016/j.entcs.2009.03.021>.
- [35] The Helium Team. *Helium Github Repository*. <https://github.com/Helium4Haskell/helium>. 2017.
- [36] Zvonimir Pavlinovic, Tim King, and Thomas Wies. “Practical SMT-based Type Error Localization.” In: *ICFP*. 2015, pp. 412–423.